PRINCIPLES OF INSTRUCTION-LEVEL DISTRIBUTED PROCESSING

BY

PIERRE M. SALVERDA

BSc Hons, University of the Witwatersrand, 1996
MSc, University of the Witwatersrand, 1998

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Craig Zilles, Chair
Professor Sarita Adve
Professor Vikram Adve
Professor Steven Lumetta
Professor Sanjay Patel

Abstract

Instruction Level Distributed Processing (ILDP) is a microarchitectural technique that distributes execution, at the granularity of individual instructions, among a number of small, independent processing elements (PEs). In aggregate, it thereby matches the execution resources of a large-window, wide-issue superscalar, but it circumvents the clock-, thermal- and power-related problems that are encountered when scaling conventional (monolithic) designs to the same dimensions. However, its distributed mode of operation introduces a number of constraints on the ability to dynamically find and exploit instruction-level parallelism (ILP). Though a very large body of research has sought to overcome those constraints, no study has, to date, been able to demonstrate instructions per clock (IPC) performance close to that of an equivalent monolithic machine. Neither, however, has any study been able to conclusively demonstrate that IPC losses must *necessarily* be incurred.

This dissertation undertakes a detailed exploration of ILDP. It seeks to understand how, and to what extent, a distributed mode of operation limits the *inherent* performance potential of a variety of designs. Where potential is found to be good, it further seeks to develop a set of practical schemes for exploiting that potential to the fullest. Its main contribution is to adopt a more general, principled approach than prior work in this area, and to thereby expose a hitherto unknown set of very basic factors at work in ILDP machines. The practical schemes it develops, which derive directly from an understanding of those basic factors, yield the best performance figures for ILDP machines that have been published to date.

The ILDP design space is divided into two parts, the first comprising machines that distribute execution among *in-order* PEs; the second, among *out-of-order* PEs. These two

classes, called, respectively, the *laned* and the *clustered machines*, have very different performance potential. The laned machines, though appealing from a clock and power point of view, are fundamentally limited in terms of their ability to exploit ILP. This is because their out-of-order execution capabilities, being now reduced to "slip" among their in-order PEs, are too coarse-grained to deal effectively with the complex and irregular manner in which ILP presents itself in the instruction stream. By means of an abstract model of laned machine operation, it is shown that matching the IPC performance of a monolithic machine demands either an impossibly complex mechanism for distributing instructions among PEs, or so many PEs that communication among them would overwhelm the machine.

The clustered machines, by contrast, are shown to be inherently capable of matching monolithic machine performance, the penalties imposed by distributed execution notwithstanding. Key to exploiting that potential is knowledge of the *critical path* through a program. This can be used to achieve a judicious allocation of execution resources to instructions, with performance-critical instructions being shielded from the distributed machine's execution constraints; only the least important instructions, which can tolerate some delay, need be exposed to those constraints. This dissertation develops several novel critical path-aware schemes, and shows that they can deliver performance that is within a few percent of a monolithic machine. It further shows that many aspects of those schemes are stable, both within and across runs of a program, a property which lends them to implementation in a static (offline) context.

Acknowledgments

Computer architecture, being as much a craft as it is a science, is learned not only through study of the literature, but also through observation of others more skilled in its practice. In this respect, I was most fortunate to have been apprenticed to so remarkably good a master as Prof. Craig Zilles. Throughout my studies, I have closely observed, and attempted to emulate, his approach to research. If I have met with any success in my work, it is without doubt due to the profound influence he has had on my thinking. I want to thank him especially for having enough faith in me to let me pursue the problems I found most interesting. I hope that I have, in the process, done my master proud.

I am also indebted to each of my committee members for their guidance and assistance, not always on matters technical. Prof. Vikram Adve and Prof. Sanjay Patel, in particular, have been involved in my work from the very beginning. I also thank Prof. Sarita Adve and Prof. Steven Lumetta, whose feedback at the early stages of my work on in-order distributed designs motivated me to dig deeper, to discover the more fundamental principles at work. Though not members of my committee, I want also to thank Prof. Matt Frank, Dr. John Sias and Prof. Kazuto Tominaga for their mentoring and encouragement.

I count myself fortunate to have been able to interact with so many first-class PhD students at UIUC. Above all, I would like to thank fellow research group members Lee Baugh and Naveen Neelakantam, who have shared with me the bulk of this long journey. I think we have observed in one another the many subtle intellectual and personal changes that accompany one's progress through the PhD program, and it has been a privilege to share that experience with them. In terms of our political and philosophical views, the three of us

constitute an eclectic mix indeed. Our debates at the Blind Pig Company have been commensurately lively! I sincerely hope that the friendships we have forged over the last five years will remain strong and lasting, in spite of our now diverging paths. Other PhD students with whom I have had fruitful interactions are too numerous to name, but I would like nevertheless to mention explicitly Mayank Agarwal, Luis Ceze, Brian Greskamp, Kshitiz Malik, Nicholas Riley, Sam Stone and Karin Strauss, to all of whom I owe thanks for their feedback on and assistance with various aspects of my work.

Finally, I want to thank my family for their unwavering support over the past six years, the many miles separating us notwithstanding. Most importantly, I thank Nicole for making this all possible, in so many different ways.

Table of Contents

List of Tables

List of Figures

Preface

To date, performance in general-purpose microprocessors has been extracted from two principal sources: frequency (clock speed) and fine-grained instruction-level parallelism (ILP). In the past, frequency improvements derived from both scaling of the process technology and from steady increases in the depth of the processor's pipeline. More recently, the latter effect has dissipated, due mainly to power and thermal constraints, but also because profitable pipeline depth is bounded by overheads that are inherent to pipelining. Instruction-level parallelism, the second source of performance improvement, derives not from the hardware, but from programs themselves. ILP arises because not all instructions in a sequential program are dependent on their predecessors; some can be processed independently of one another. ILP offers two complimentary opportunities for improved performance. First, independent instructions can be executed in parallel with one another, increasing average instruction throughput and, thereby, reducing overall runtime. Second, instructions can be re-ordered to permit execution of useful work "in the shadow" of unpredictable events like cache misses, thereby hiding their latency.

Various schemes have been proposed for exploiting ILP, but by far the most successful has been the *dynamically-scheduled superscalar processor*, now ubiquitous in general-purpose client machines. These processors mine ILP from the instruction stream dynamically, an approach that renders them versatile enough to sustain very good instructions-per-clock (IPC) performance across a wide range of application domains, in the presence of highly variable memory access latencies, and in spite of complex control flow patterns. This dissertation is concerned with design challenges facing dynamically-scheduled super-

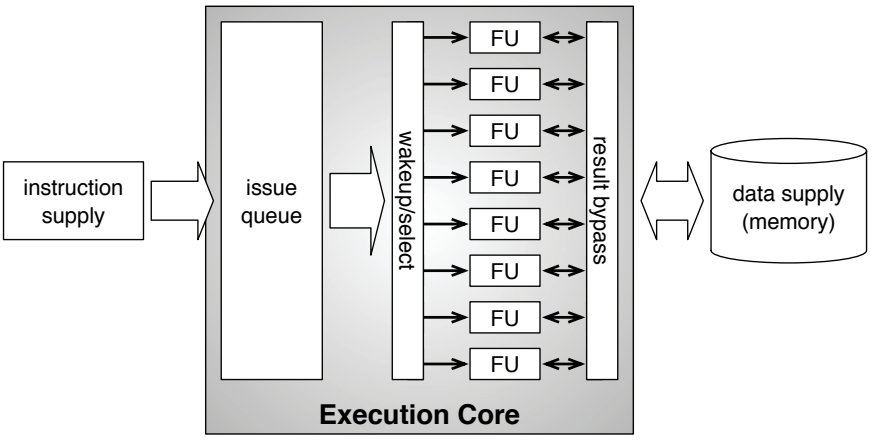


Figure 1. A typical out-of-order superscalar machine. Each cycle, a dynamic scheduler (the wakeup/select logic) finds and issues data-ready instructions from the issue queue. The 8 functional units facilitate 8-wide superscalar execution.

scalars and, specifically, with techniques that attempt to simplify the machine's execution logic by reducing its dynamic scheduling capabilities in some way. The sections that follow provide context for that work and motivate the specific solutions it explores.

Out-of-order execution

Figure 1 depicts a dynamically-scheduled superscalar machine. At a high level, the machine comprises three parts: an instruction supply, a data supply and an execution core. The task of the instruction supply component is to deliver a (speculative) stream of instructions into the execution core. The latter then processes those instructions according to a dynamically-determined execution plan, at times initiating interaction with the data supply component to read from and write to the memory system. Such a machine's performance is determined by a number of factors. Among them are the average instruction supply rate, the average latency of memory operations, the amount of ILP available in the instruction stream, and the ability of the execution core to find and exploit that ILP. The principal focus of this dissertation is on the last of these.

Within the core, processing occurs as per the dataflow dependences embedded in the

instruction stream. Each cycle, a dynamic scheduler (the wakeup/select logic in the diagram) inspects the contents of the issue queue, looking for instructions whose operands are ready, and for which execution resources are available. It then issues those instructions to the functional units for execution, in the process rendering their dataflow dependents, if any, eligible for issue in subsequent cycles. A result bypass network facilitates rapid issue of those dependent instructions by permitting them to receive their operands directly from the recently-issued producers. Within the confines of its execution core, then, the machine is generally able to execute instructions subject only to the constraints imposed by program dataflow. The flexibility thereby achieved allows the machine to dynamically respond to unanticipated events like cache misses and control flow misspeculation, and so to partially or wholly hide their effects in the shadow of useful work. It is this capability that underlies the success of the dynamically-scheduled machines. It is, unfortunately, also the principal constraint preventing expansion of these designs to further exploit ILP.

Pursuit of more ILP involves scaling an out-of-order design in two, largely orthogonal, dimensions. The first is the machine's *depth* — its window size. A larger window improves the ability to find ILP by permitting the machine to maintain a larger set of in-flight instructions, and so to expose more independent instructions to the scheduling logic. The second is the machine's *width* — its issue and execute bandwidth. A wider issue capability improves the machine's ability to exploit ILP when it is available, since it increases the rate at which independent instructions can be extracted from the issue queue. Expanding a machine's size in either of these dimensions runs head on into problems on a number of fronts.

Technology scaling. Continued scaling of the process technology has exposed wire delay [41, 61, 97], power consumption [37, 66, 80] and thermal output [13, 80] as first-order constraints on microprocessor design. Dynamically-scheduled superscalars suffer especially under these constraints because their microarchitectures — the execution core, in

particular — are replete with large, wire-limited and power-hungry parts. For example, the wakeup/select scheduling logic and the result bypass logic are both inherently wire-limited, so scaling the machine's width or depth is liable to compromise the machine's cycle time [76]. Equally, enlarging the issue queue and increasing the machine's issue width will exacerbate power consumption and power density (*i.e.* thermal) problems.

Instruction supply. A large instruction window is only beneficial to the extent that it can be filled with instructions. Though branch prediction has improved phenomenally over the past decades, even a small misprediction rate places a very severe bound on the average number of useful (*i.e.* correct-path) instructions that can be fetched before a branch misprediction drains the machine's window.[1] Lam and Wilson [57] showed that this constraint places a hard limit on the amount of ILP visible to, and hence exploitable by, a superscalar machine.

Data supply. The growing gap between DRAM latency and processor clock speed is a perennial problem in computer architecture. As this gap grows, the utility of scaling a machine's issue width drops: when memory latencies increase, the program's dataflow graph is effectively stretched, and available ILP can be serialized in the shadow of the long-latency memory operations.

The first of these problems is a fundamental one, in the sense that technology-related issues arise as a direct consequence of the complex circuitry necessitated by the out-of-order execution machinery. Instruction and data supply problems are, in this respect, somewhat orthogonal; they are present no matter what the underlying microarchitecture looks like. More to the point, any technique that can address or mitigate instruction and data supply bottlenecks (*e.g.* better branch predictors, faster memory technologies) will amplify the

[1] For example, assuming a misprediction rate of 5%, and an average conditional branch frequency of 20%, the machine can fetch only 100 instructions (20 branches) before a misprediction occurs.

utility of further scaling a dynamically-scheduled design. But doing so demands first overcoming the implementation problems faced by such machines, and so it is to this more fundamental problem that this dissertation devotes its attention.

Tackling the implementation challenges

The implementation problems faced by dynamically-scheduled machines have been the subject of intense research for well over a decade now. This dissertation focuses on *Instruction Level Distributed Processing* (ILDP) [90], a term introduced by Jim Smith to describe a general paradigm for the design of superscalar machines in an era in which concerns about wire delays, power consumption, thermal output and design complexity are paramount. Having only been introduced in 2001, ILDP postdates a very large body of research into a closely related class of machine called the *clustered superscalars* [8, 32, 49, 76, 77]. In a sense, ILDP is the culmination of all that previous work, an explicit and deliberate enumeration of a set of design principles for tackling the mounting problems posed by technology scaling. That is, ILDP is really an umbrella term for a large family of machines that share its design principles, and it is in this broad sense that the term will be used throughout this dissertation.

The principal motivating force in ILDP is the wire delay problem. Because the speed of wires has not scaled at the same rate as that of transistors, a homogeneous view of the chip — from a communication point of view — is no longer possible. Instead, the die has become a partitioned one, comprising many local regions within which communication can occur intra-cycle, but between which communication must be deemed global, requiring multiple clock cycles. This leads naturally to the notion of a partitioned, or distributed, design for the execution core. Figure 2 shows how the 8-wide machine from Figure 1 can be partitioned into four 2-wide *processing elements* (PEs). Such a design eliminates the large, wire-limited and power-hungry parts of the monolithic design, replacing them

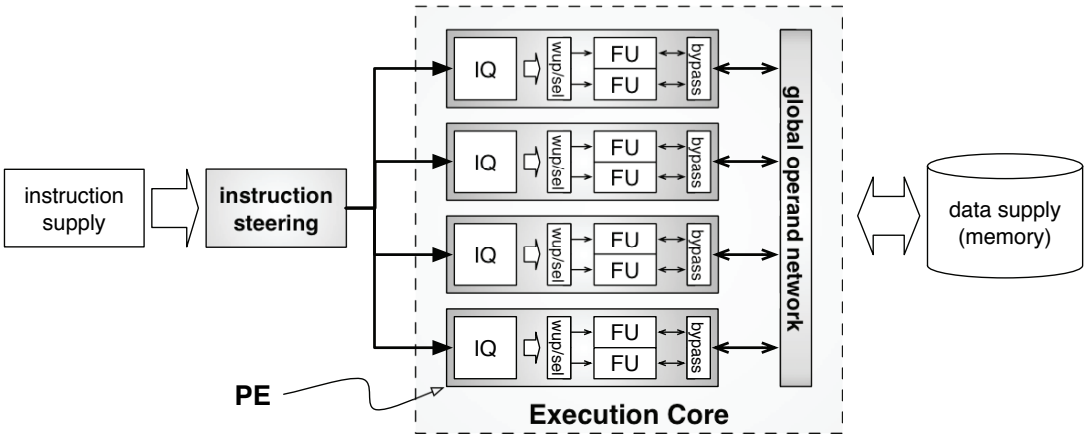


Figure 2. Instruction-level distributed processing. This design partitions the execution resources of the machine shown earlier in Figure 1 into 4 independent processing elements (PEs), each a self-contained 2-wide dynamically-scheduled execution unit. Instruction steering logic is added to the front-end of the machine to dynamically allocate each fetched instruction to the PE at which it will eventually execute. Communication of operands between PEs occurs via a global bypass network, which introduces additional latency between dependent instructions.

instead with small, replicated components. This permits localization of work at relatively simple execution units that can be engineered to be fast (because communication is local), but it maintains, in aggregate, the large window and wide issue capabilities required for aggressive pursuit of ILP. And the fact that PEs are small and replicated offers opportunity for tackling the power problem, both on the static and the dynamic front.

Of course, these implementation benefits come at a cost. A partitioned design introduces a number of constraints on instruction execution — constraints that are either not present, or present only to a very limited extent, in a monolithic design. For example, transferring values computed at one PE to consuming instructions at another now incurs a *global communication penalty*. There is now also potential for load imbalance and, as a result, for spurious *resource contention stalls* incurred when instructions must wait for a chance to execute at an over-subscribed PE. Some ILDP proposals, which distribute execution among (independent) in-order PEs, are even further limited by an *in-order issue constraint* now imposed on subsets of all instructions. In general, these constraints constitute a deviation from the ideal of a monolithic mode of operation, since each acts to prevent

instructions from executing as soon as their operands have been produced. The principal goal of this dissertation is to explore the ramifications of these changes.

Performance challenges in ILDP

The extent to which any one of the above constraints has an impact on performance is clearly dependent on how instructions are distributed among the PEs. A key objective in an ILDP machine, therefore, is to find a distribution of instructions among the PEs that mitigates, or perhaps even eliminates, their effects. This is called the *instruction placement*, or *steering*, problem, and it has been the central focus of a very large body of research. Proposals vary, ranging from static schemes implemented in the compiler [32, 47, 59, 73, 77] to fully dynamic schemes implemented in the machine's front-end pipeline, as per Figure 2 [6, 8, 12, 19, 35, 49, 76, 78, 98]; some adopt a hybrid approach involving both hardware and software components [52, 53]. Though they differ in specific details, all of these studies fall into the broad category of *dependence-based steering* policies, since all of them use dataflow dependences to govern the placement process. The guiding principle in such schemes is an intuitive one: dependent instructions ought to be collocated at a single PE to avoid global communication penalties, while independent instructions ought to be spread across PEs to alleviate resource contention. And, where PEs support only in-order execution, a dependence-based approach has the propitious benefit of subsuming a PE's in-order execution constraint with dataflow dependences: in-order issue at a PE is benign if all instructions sent there are part of a chain of dependent instructions.

Unfortunately, dataflow relationships, alone, are not a sufficient basis for a steering policy. Even a cursory examination of dynamic dataflow reveals that its shape, and the distribution of ILP within it, is far from regular. For example, dataflow frequently diverges (an instruction can have more than one dependent), so attempting to distribute instructions based solely on their dataflow relationships is liable to direct too much work at just one PE;

the ensuing resource contention stalls will serialize the available ILP. This line of thinking led to a prevailing view that resource contention and global communication are opposing forces, and hence that the best performance will be obtained when the right balance between the two is obtained. Indeed, some researchers proposed metrics for quantifying workload (im)balance explicitly, and used this data to dynamically adjust the behavior of standard dependence-based placement rules [1, 18–20, 39, 78].

Couching the problem in terms of a trade-off between locality and workload balance is intuitively appealing, and it lends itself to practicable implementation mechanisms for dynamic steering logic. But steering policies developed along those lines have failed to deliver good performance. In fact, most of the studies conducted in the 1990s reported substantial IPC losses relative to a resource-equivalent monolithic superscalar, leading ultimately to a pervasive view in the field that such IPC losses are inevitable in a partitioned design. Researchers have therefore tended to invoke gains on the clock rate and power front as the principal benefits of ILDP; efficacy at exploiting ILP is not a selling point. That view changes, however, if the instruction placement problem is tackled from a different perspective, one that abandons the use of *aggregate* measures of global communication and workload imbalance as targets for optimization.

The problem with aggregate metrics is that they are homogenizing. Their use embeds a tacit assumption that all instructions contribute equally to performance, and hence that their optimization will translate into an overall improvement in performance. However, a program's runtime — its *critical path* — is determined by only a subset of the instructions it executes [35, 98]. Techniques that use aggregate metrics, therefore, are too egalitarian: they are liable to target instructions that have no bearing on performance, which constitutes wasted effort, and they can fail to give sufficient attention to instructions that directly impact performance, which constitutes lost opportunity.

Two factors combine to render awareness of a program's critical path important in the context of ILDP. The first is the set of constraints that an ILDP machine imposes on instruc-

tion execution. Whereas instructions in a conventional machine are able to execute as soon as their operands have been produced, and hence at the peak rate allowed by the dataflow within the machine's window, some fraction of instructions in an ILDP machine will be delayed beyond their data-ready time by ILDP-specific constraints.[2] The second factor relates to the fact that not all instructions reside on the critical path; many — the majority, in fact — can be delayed without any deleterious effects on runtime [33]. Taken together, these observations imply that the performance of an ILDP machine will suffer only to the extent that its execution constraints are imposed on the few important instructions that affect performance, and also to the extent that unimportant instructions are able to tolerate those constraints. This points to an opportunity. If execution resources can be managed judiciously, so that only the non-critical instructions incur ILDP-specific penalties, then it might be possible to sustain good performance in spite of those penalties.

Several researchers have noted and exploited this opportunity. In the context of statically-scheduled clustered VLIW machines, for example, compilers use criticality to give priority to the longest dataflow chains during their scheduling and cluster-assignment passes, thereby aiming to ensure that critical instructions on long dataflow chains do not incur global communication penalties [27, 47, 59, 72, 73]. In dynamically-scheduled machines, however, the potential for unanticipated long latency events like cache misses and branch mispredictions renders static properties of dataflow less useful as a measure of criticality. In general, execution in a dynamic machine comprises a number of potentially-critical paths, their interplay being determined dynamically by events at runtime. Schemes have therefore been devised for incorporating a critical path predictor into the pipeline, the output from which is used to guide instruction steering and scheduling decisions implemented by the hardware [35, 98].

[2] Actually, conventional superscalar machines are not entirely free from these constraints. Resource contention stalls can arise when a burst of ILP introduces more data-ready instructions into the window than the issue logic is able to remove from it. But such cases are relatively uncommon, especially in wide-issue machines. More importantly, when such stalls do arise, the machine is operating at, or close to, its peak performance, in which case the problem becomes moot.

Research goals

Though prior work has made enormous strides in overcoming the performance constraints imposed by various ILDP designs, particularly where instruction criticality has been brought to bear on the problem, no study has yet demonstrated performance that might be deemed close to that of a resource-equivalent monolithic design. Given that modest superscalars are already being built by commercial chip vendors, and that IPC gains from scaling to more aggressive superscalar dimensions are already modest (because of limits imposed by instruction and data supply problems), the case for an ILP-inefficient design — its implementation benefits notwithstanding — is not a very compelling one. The challenge taken up in this dissertation is to determine if the ILP capabilities of ILDP designs can be improved.

As noted earlier, prior work has explored a variety of strategies for overcoming the performance problems posed by ILDP designs. Each of those studies, however, has tended to focus on one or more instruction steering heuristics, and has evaluated those heuristics for just one point in the ILDP design space. Though a restricted focus is often a pragmatic necessity, the unfortunate consequence is a plethora of steering policies and a disparate collection of empirical evaluations. This renders comparisons between studies very difficult, which, in turn, means there is a lack of global perspective on this design space — on its as yet unconfirmed potential to deliver good performance, and on the best means to go about getting that performance in practice (should it indeed exist). This dissertation aims to address the problem by attacking it from the bottom up, seeking first to understand the basic capabilities (from an IPC point of view) of the underlying microarchitecture. It seeks, first and foremost, to identify the most important factors affecting the performance potential of (a variety of) ILDP designs. Insights thus gained facilitate the second major objective, which is to develop practicable mechanisms for exploiting performance potential where it is found. Hence the following high-level research goals.

Analysis: performance in principle. The objective is to understand to what extent programs admit shifting of ILDP-induced performance constraints from critical onto non-critical instructions; and, equally, to what extent the constraints inherent to the underlying microarchitecture lend themselves to being selectively imposed on a subset of instructions. Specific questions tackled include the following.

- *What is the inherent (idealized) IPC potential of a given partitioned-core design?* Quantifying this potential is important for a number of reasons. First, it serves to justify a more detailed analysis of a given machine: unavoidably high IPC losses would call into question the merit of trying to implement such a design. Second, a best-case IPC loss establishes a lower bound on the clock rate improvements the design would have to achieve in order to break even in terms of net performance. Finally, the ideal IPC performance potential can serve as a reference point — a gold standard — against which real implementations of the machine can be measured.

- *What are the first-order effects that determine performance of a given machine?* The performance of even monolithic out-of-order machines is determined by a complex set of interactions between a program's dynamic dataflow patterns and the underlying hardware resources that execute them. How does an ILDP design change those interactions? Are any new factors introduced; are some dampened; amplified? Understanding, in particular, the role played by dataflow permits delineation of the set of programs — or, better yet, program characteristics — to which a given ILDP design is well suited; and, equally, those to which it is not.

Design: performance in practice. Where potential has been identified, the next challenge taken up is to determine how critical path detection and prediction techniques can be used to realize that potential in practice. Specifically:

- *What policies are needed for managing execution resources in such a way that the constraints imposed by a distributed execution core are mitigated?* Policies derive

directly from an understanding of the basic factors that determine performance; they are designed to target specifically those performance-degrading effects that arise in the context of a given ILDP design.

- *What mechanisms might best implement those policies, and can they be engineered with reasonable complexity?* Any implementation mechanisms demanded by an ILDP machine ought not to introduce more complexity than is removed by partitioning the execution core in the first place. That said, where hardware complexity seems unavoidable, there might be potential to migrate some of it into software. This will be the case if policy logic is not inherently dynamic, in the sense that some, or all, of the decision making is statically predictable.

Looking forward: CMPs

With the advent of chip multiprocessors (CMPs), and the corresponding shift away from ILP-centric design toward pursuit of more coarse-grained thread-level parallelism (TLP), it might appear that the need for a detailed study of ILDP has been eclipsed. However, the results of this dissertation will have particular relevance in the new CMP era. This is because single-thread performance will remain important, and TLP is no panacea: not all programs are amenable to coarse-grained parallelization, be it via automatic or manual techniques. Moreover, even parallel programs have sequential parts, and these will bound the performance benefits of parallelization [40]. It therefore seems likely that two main classes of program will dominate future workloads. The first is the set of massively-parallel, computationally-intensive programs in areas such as graphics, physics, signal processing, and recognition, mining and synthesis (RMS). The second is the very large body of remaining applications that resist parallelization. Programs in the first category will clearly thrive in the TLP-centric regime of future CMPs: parallel codes benefit from a wealth of narrow, perhaps multithreaded, cores of modest clock speed, with wide SIMD functional

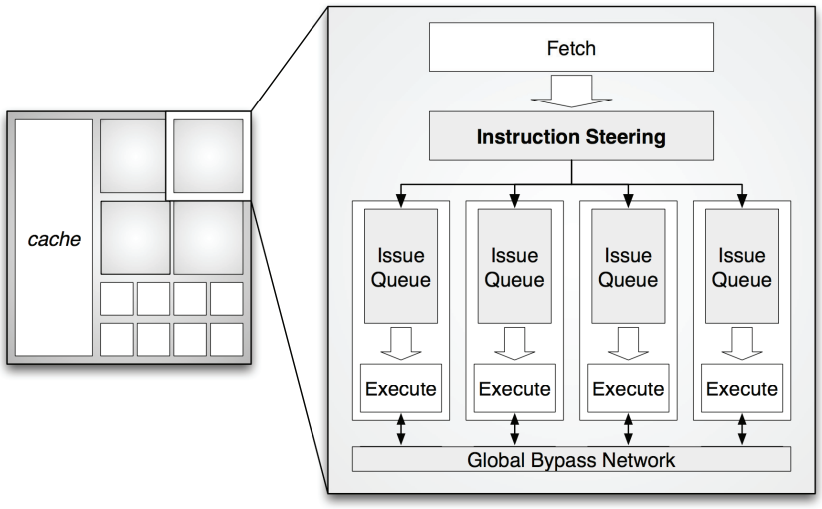


Figure 3. Heterogeneous CMP. One possibility for future chips involves embedding a number of cores on a single die, not all of which share the same microarchitecture. The diagram shows one possible scenario in which the chip comprises 8 small cores to deal with parallelized code, plus 4 large and powerful ILP processors for sustaining high performance on sequential code. For the latter cores, an 8-wide ILDP design is an appealing option.

units and a high-bandwidth memory system. By contrast, the sequential codes in the second category need a single, high-frequency, wide-issue, aggressively-speculative processor with a low-latency memory system.

Reconciling these conflicting hardware requirements is challenging. One well-studied proposal is the heterogeneous CMP, an example of which is depicted in Figure 3. This machine comprises a small number of aggressive dynamically-scheduled superscalar cores, and a larger number of small, efficient, throughput-oriented cores. It is to the former that the results of this dissertation apply. These will have to deliver good performance on non-parallel code, both through exploitation of ILP and through sustaining high clock rates. But they will have to do so within increasingly tight constraints on complexity, area and power. These are precisely the constraints to which ILDP designs lend themselves.

An alternative strategy for catering to sequential workloads on a CMP is shown in Figure 4. In this case, a homogeneous many-core CMP is extended to support on-demand aggregation, or fusion, of small processor cores into a large out-of-order uniprocessor. The principal benefit of this approach is that it requires the design of just one simple core,

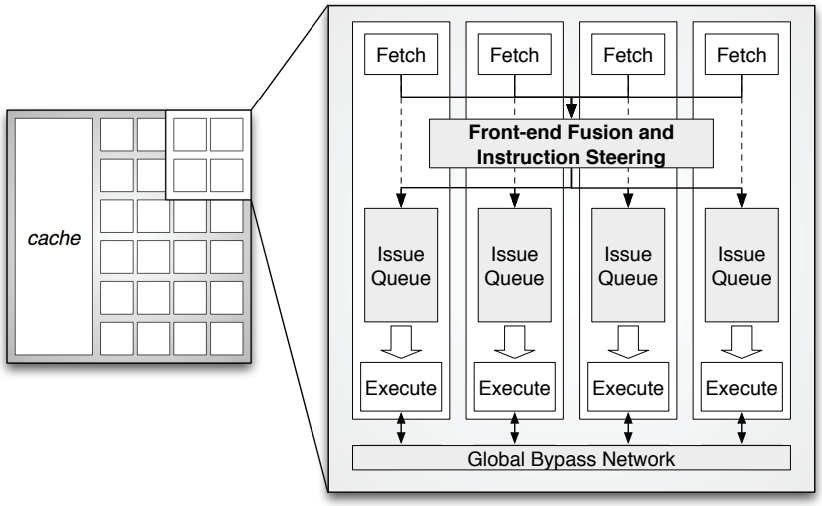


Figure 4. Homogeneous CMP with core fusion. The diagram shows an alternative to the heterogeneous design shown in Figure 3. In this case, the chip embeds 24 small cores, all exactly the same, but all of too modest a dimension to serve sequential codes well. The CMP therefore supports on-demand fusion of (subsets of) its small cores into a single, large out-of-order processor capable of extracting ILP in sequential codes. That composite execution core is exactly analogous to the partitioned ILDP core shown in Figure 3.

yet is flexible enough to cater to a wide range of TLP and ILP in the workload. As the figure shows, a fused-core design is, in principle, not very different from a partitioned-core design; they are really two sides of the same coin. Hence, the results and observations presented in this dissertation will have bearing on future research into fused-core designs. Of particular importance for this area is this dissertation's exploration of ILDP designs comprising in-order PEs. All of the many-core CMPs being produced or planned by chip vendors comprise in-order cores only. The ability to fuse such cores to achieve out-of-order performance on single-threaded programs constitutes the holy grail of modern computer architecture. This dissertation will show, however, that there are fundamental limits on the performance that can be extracted from such designs.

Chapter 1

Introduction

In this dissertation, I explore *Instruction Level Distributed Processing* (ILDP) as a means for tackling the clock and power problems suffered by conventional dynamically-scheduled superscalar designs. To varying degrees, the different ILDP designs studied here introduce constraints on the machine's underlying execution model — deviations in some way from the ideal of dataflow-oriented execution, where the only constraints on execution are dataflow dependences. Those deviations offer benefits from an implementation point of view, but all of them change in some way the machine's inherent ability to dynamically find and exploit ILP in the instruction stream. The principal objective of this work is to quantify those limits, and thus to identify points in the ILDP design space that are likely to be profitable avenues for further investigation. Furthermore, since performance in principle does not necessarily translate directly into performance in practice, a second, but equally important, objective is to develop mechanisms necessary for realizing the hardware's underlying potential. And since these designs are being explored for the purposes of simplifying the hardware, an important criterion is to ensure that the proposed mechanisms are indeed complexity-effective, in the sense that they do not re-introduce the complexity they aim to remove in the first place.

In this introductory chapter, I describe the organization of this dissertation, summarizing its main results and conclusions. I also provide some background information on the principal empirical and analytical techniques that are used throughout my work. So as to delineate the scope of this dissertation, I also describe the main limitations of this work, particularly the set of implementation challenges not solved by ILDP. First, however, I

describe below the overall design space to which this work is devoted.

1.1 The microarchitectural landscape

In very general terms, this dissertation concerns itself with three principal types of machine: conventional superscalar designs and two types of ILDP design. I distinguish these machines based on the abstract notion of an underlying *execution model* implemented by the hardware. I describe these models in Section 1.1.2. I make no attempt to do so rigorously, nor do I claim that the three models I enumerate constitute a definitive partitioning of the design space. Rather, I use the execution models merely as a framework in which to reason about, and to compare the behavior of, the different types of machine in which I am interested. Moreover, each model distills the key capabilities — more importantly, the key constraints — that ultimately distinguish machines in terms of their inherent ability to dynamically find and exploit ILP. Enumerating these *explicitly* in the form of simple, concise — albeit informal — execution models is therefore a useful exercise in itself. Before I describe these models, I must make a brief digression at this point to clarify some of the terminology I will use throughout this dissertation.

1.1.1 Terminology

The literature is not always consistent in its use of a number of terms. In the interests of precision and clarity, I shall commit to the following conventions throughout this dissertation. First, machines targeting ILP can be broadly divided into two classes: the *static* and the *dynamic* machines, also called the *in-order* and the *out-of-order* machines, respectively. I will use the term static (in-order) to refer to any machine in which execution order is statically fixed (encoded in the program binary). Such machines rely on the compiler to expose ILP to hardware. Early superscalars such as the Alpha 21064 [26] and the 21164 [9] fall into this category, as do the VLIW and EPIC machines [23, 62]. In the

dynamic (out-of-order) machines, by contrast, execution order is determined at runtime, being controlled now by hardware and the dataflow dependences hardware infers from the instruction stream. Microarchitectural techniques such as register renaming, memory disambiguation and dynamic scheduling characterize such machines. The MIPS R10000 [100] and the Alpha 21264 [50] are good examples, as are more recent commercial designs such as Intel's Centrino and Core 2 and AMD's Athlon and Opteron.

I make these distinctions explicit because, in the context of ILDP, the line between static and dynamic can become somewhat blurred. As I will show below, some ILDP machines do not support dynamic scheduling at all. But, because their various execution units operate in a decoupled manner, instructions can become reordered at runtime. Execution order is therefore not determined by the program binary, requiring that the machine dynamically manage inter-instruction dependences to ensure correct operation. I therefore qualify such designs as dynamic. In fact, all of the work I present in this dissertation is concerned exclusively with the dynamic machines.

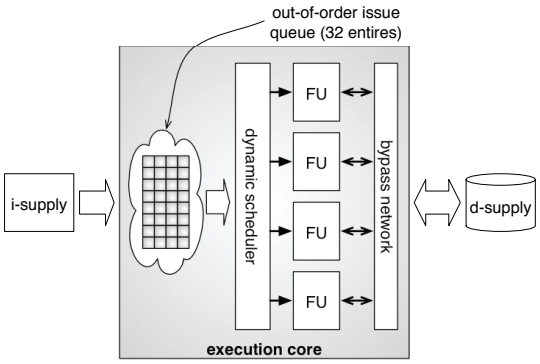
I want to again be explicit about my use of the term *ILDP*. I use this, as did Smith [90] when he introduced the term, to refer to any machine in which processing is distributed, at the granularity of individual instructions, among some number of independent and comparatively simple *processing elements* (PEs). I stress this because previous work explicitly aligning itself with ILDP terminology has become synonymous with a more specific design approach, one which places an emphasis on clock rate over instructions per clock (IPC) performance, promoting, as a result, designs comprising very simple, very fast in-order PEs [52, 53]. Such designs clearly fall within my scope, but so too do machines supporting out-of-order execution at their PEs. Equally, I do not specifically assume that high clock speed — or low power consumption, or design simplicity — are the primary design goals (though they may well be). I also include within the compass of ILDP the recently proposed schemes for *fusing*, or *aggregating*, small cores in a CMP [44,51]. Though these machines introduce a number of additional hardware components to manage other-

wise disjoint processors as a single unit (particularly in the front-end and memory system), they share with the other ILDP machines the same underlying (instruction-level) distributed execution model.

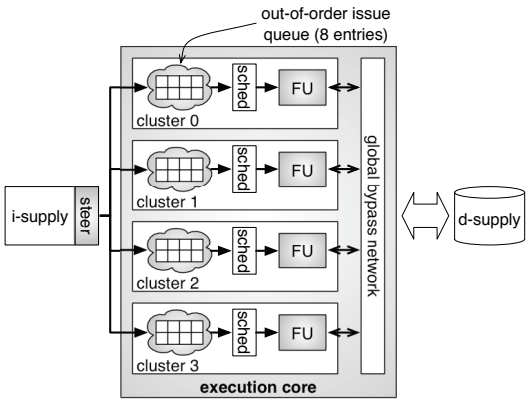
1.1.2 Out-of-order execution models

My focus in this work is on a dynamic machine's execution core. More specifically, it is on the inherent capabilities of a given design to find and exploit ILP in the instruction stream. Understanding and reasoning about those capabilities is facilitated by describing the *execution model* supported by that design. Below I describe the three models on which all of my work is centered. The first, which is representative of conventional out-of-order designs, might be described as *monolithic*, in the sense that its main execution resources comprise large, centralized structures. The remaining two, which are implemented by ILDP machines, are, by contrast, *distributed*, with execution being managed separately at two or more independent structures. These distributed models are characterized by their introduction of a *local* (intra-PE) and a *global* (inter-PE) dimension to execution, a distinction not present in the monolithic model.

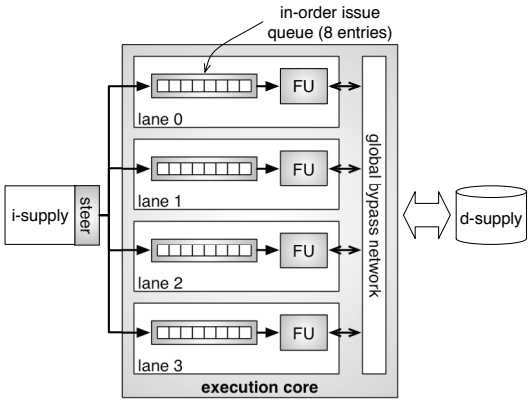
The dataflow-oriented execution model. Most commercial machines, the MIPS R10000 being perhaps the archetypal example, implement what I call a *dataflow-oriented execution model*. This is depicted in Figure 1.1(a). The machine operates by maintaining a window of in-flight instructions, each cycle inspecting the contents of that window in search of instructions that are eligible for execution. This is achieved by effectively computing a dynamic dataflow graph for all instructions in the issue queue, which holds that portion of the window yet to be executed. Each cycle, the dynamic scheduler selects, from among the roots of that dataflow graph, the set of instructions that will be executed next. Instructions thus issued to the functional units are then pruned from the dataflow graph, thereby exposing their dependent instructions as the new dataflow roots, and hence rendering them eligi-



(a) Dataflow-oriented



(b) Distributed dataflow-oriented



(c) Slip-oriented

Figure 1.1. Out-of-order execution models. Diagram (a) abstractly depicts a conventional out-of-order superscalar machine. Its monolithic execution core is responsible for maintaining a window of in-flight instructions from which dynamic scheduling logic selects some number of data-ready instructions for execution each cycle. The dynamic behavior thereby achieved is key to the machine's ability to respond to and, to some extent, mitigate the performance-degrading effects of branch mispredictions in the front-end (i-supply) and of cache misses in the memory system (d-supply). Diagrams (b) and (c) show the same net execution resources now distributed among out-of-order and in-order PEs, respectively. In both machines, dynamic steering logic is added to the front-end to manage the distribution of instructions among the PEs, and a global bypass network is added to the execution core to distribute operands among the otherwise disjoint units.

ble for execution in subsequent cycles. At the same time, newly fetched instructions are inspected for dependences on in-flight instructions, and inserted as leaves into the graph. Because instructions enter and leave the window in the order dictated by the instruction stream, the machine adheres to a sequential execution model. But, within the confines of its window, it operates according to a dataflow-oriented model, where instruction execution is constrained by dataflow dependences only, not by sequential ordering constraints. This

very flexible mode of operation is key to the machine's ability to dynamically respond to, and to partially hide the effects of, performance-degrading events like cache misses and branch mispredictions. Throughout this dissertation, I adopt the view that this dataflow-oriented model is the ideal to which any ILDP design strives.

The distributed dataflow-oriented execution model. The *clustered superscalars* studied in the late 1990s [8, 32, 49, 76, 77] and the recently proposed schemes for aggregating small out-of-order cores in a CMP [44, 51] are all examples of what I call *distributed dataflow-oriented execution*. As Figure 1.1(b) shows, machines in this class distribute execution among *dynamically-scheduled* PEs. That is, each PE is a small, self-contained out-of-order execution core, and so achieves a degree of dataflow-oriented execution. But the fact that they are disjoint constrains the machine in two respects.

1. *Communication.* Transferring values computed at one PE to consuming instructions located at another incurs a *global communication penalty*. This effectively lengthens any dataflow chains that cross PE boundaries, and so prevents back-to-back issue of dependent instructions.

2. *Contention.* With a distributed design comes the potential for load imbalance, and hence for spurious *resource contention stalls* as data-ready instructions wait for a chance to execute at one PE while other PEs remain idle.

Both of these constraints constitute a deviation from the ideal of dataflow-oriented execution, since each acts to prevent instructions from executing as soon as their operands have been produced. So as to remain consistent with previous work in this area, I will refer to machines implementing the distributed dataflow-oriented model as the *clustered machines*; each PE will accordingly be referred to as a *cluster*.

The slip-oriented execution model. Figure 1.1(c) depicts *slip-oriented execution*, an alternate form of instruction-level distributed processing. In addition to introducing the afore-

mentioned communication and resource contention constraints on execution, this further deviates from dataflow-oriented execution by imposing a *local in-order issue constraint*: subsets of the instruction window must issue in program order. Out-of-order execution is still possible, but it can occur now only in a coarse-grained manner, being reduced to "slip" among the (independent) in-order PEs. The dataflow-oriented model, by contrast, effectively supports slip at a very fine granularity: any instruction can execute out-of-order with respect to any other. One of the first examples of these machines, though not properly considered part of the ILDP family, is the dependence-based scheduler proposed by Palacharla *et al.* [76]. This partitions the issue queue into a number of FIFO buffers, thereby constraining subsets of in-flight instructions to in-order execution. More recently, slip-oriented machines were promoted as a complexity-effective trade-off between clock rate and IPC [52, 53]. Throughout this dissertation, I will refer to machines implementing slip-oriented execution as the *laned machines* and to the individual PEs as *lanes*.

I again want to emphasize that the above characterizations are merely a useful conceptual framework in which to work; they do not constitute a definitive breakdown of the microarchitectural landscape. In particular, no commercial design looks exactly like the monolithic machine depicted in Figure 1.1(a). Most divide their execution core into distinct components based on instruction type — separate integer and floating point issue queues, for example. Though not truly monolithic, I do not view these machines as implementing a distributed execution model for the simple reason that instruction dispatch has no choice in terms of where instructions are sent: floating point instructions, for example, can be dispatched only to the floating point issue queue. A machine like the Alpha 21264 [50], however, is different in this respect. Indeed, it is frequently referred to as a clustered machine in the literature. Nevertheless, its issue queue remains a monolithic structure, with only the functional units and register file being distributed. Though it embodies a number of ILDP design ideals, the fact that its issue queue remains monolithic means its wakeup

logic, which is the principal source of implementation complexity in dynamic schedulers, cannot easily scale with increasing issue queue size. As such, the Alpha 21264 tackles most, but not all, of the implementation challenges that are the focus of this dissertation.

1.2 Dissertation outline and summary of results

The two distributed execution models identified above differ fundamentally in terms of their inherent performance capabilities. I therefore divide this dissertation into two parts, the first devoted to slip-oriented execution (the laned machines), which is the more constrained of the two; the second, to distributed dataflow-oriented execution (the clustered machines). Below, I summarize my main results and conclusions for each.

1.2.1 The laned machines

Part I of this dissertation is devoted to the laned machines, whose slip-oriented execution model sets them apart from the clustered machines. As I noted above, this model constrains subsets of in-flight instructions to execute in fetch order, permitting out-of-order execution only among those subsets. Though it offers enormous benefits from an implementation point of view, I will show that the slip-oriented execution model is inherently limited in terms of its ability to exploit ILP. Specifically, I will show the following.

Revisiting dependence-based scheduling. In the late 1990s, Palacharla *et al.* [76] proposed a complexity-effective implementation of dynamic scheduling logic in which the issue queue is partitioned into a number of FIFO buffers, thereby constraining subsets of all buffered instructions to issue in program order. Their scheduler is, in this respect, exactly analogous to slip-oriented execution, though it does not impose a global communication penalty (the functional units are not partitioned). As such, it serves as a good starting point for understanding just the impacts of the partially in-order mode of operation

that characterizes slip-oriented execution. The very good performance results published for that design would seem to indicate that this constraint is largely benign. However, I show in Chapter 3 that those results are somewhat misleading. The proposed scheduler's surprisingly good performance relative to a monolithic machine can be ascribed to the presence of bottlenecks outside of the execution core, which hide an underlying problem with dependence-based scheduling and, ultimately, with slip-oriented execution.

Practicable potential. The performance achievable by the aforementioned scheduler is determined entirely by the dependence-based steering policy used for steering instructions among its FIFO buffers. I show in Chapter 4 that improving on the results achieved by that policy is fundamentally hard. Ultimately, properties of dataflow that are to blame. Specifically, sustaining a given IPC generally involves execution of instructions from a much larger number of simultaneously active dataflow chains — an artifact of the non-uniform shape of dataflow and of the varying distribution of ILP within it. When mapped onto a partially in-order execution model, this general property, in turn, demands either that the active dataflow chains be carefully interleaved among the available lanes, or that enough lanes be available for each chain to reside at a distinct lane. Using an abstract model for reasoning about the performance of these machines, I show that the former option is fundamentally hard, in the sense that it necessitates instruction steering hardware that would be even more complex than dynamic scheduling itself. The latter option would demand so many lanes that the machine would be overwhelmed by global communication penalties.

Overall, these results have particular relevance in the modern CMP era. The holy grail microarchitecture for future client machines, which will have to be able to deal effectively with both sequential and parallel programs, is a CMP comprising a "sea" of small, power-efficient in-order cores (ideal for TLP-rich codes), but which also supports on-demand

fusion of those cores to synthesize a large, out-of-order processor (ideal for exploiting ILP in sequential codes). Such a machine model introduces a number of new performance constraints, particularly in the front-end and in the memory system, but, within the confines of its (aggregated) execution core, is no different from the laned machines to which the above results apply. In fact, because my observations about fundamental performance limits derive from an idealized view in which a global communication penalty is absent, let alone the additional overheads that will inevitably arise in the context of a fused design, the prospects of a fused machine matching the IPC of even modest monolithic designs are not good. This this does not mean that fusing of in-order cores is entirely out of the question. If efficiency (IPC/Watt) is adopted as a metric, for example, such designs might become very appealing. However, this result does mean that single-thread performance will necessarily suffer — and rather severely — if in-order fusion is adopted as the sole means of pursuing single-thread performance.

1.2.2 The clustered machines

From a performance point of view, the clustered machines are much more compelling than the laned machines. This is not surprising given that the clustered machines depart much less from the dataflow-oriented execution model than do the laned machines. Specifically, their distributed dataflow-oriented execution model only constrains execution through the imposition of a global communication penalty and resource contention stalls. In Part II of this dissertation, I explore the ramifications of those constraints in detail, presenting the following main results.

Idealized potential. Using an idealized steering infrastructure, which can oracularly identify those instructions that have the most influence on overall performance (*i.e.* that are most critical), I show in Chapter 6 that program dataflow is remarkably tolerant of the constraints imposed by a distributed dataflow-oriented model. Even with communication penalties of

up to 4 cycles (a pessimistic number), I find that the IPC potential of a variety of clustered machines is consistently within 5% of that of a resource-equivalent monolithic design. This is an important result for two reasons. First, it proves that dataflow readily lends itself to being processed in a distributed fashion. Specifically, critical dataflow chains are generally narrow enough to execute at even the smallest PEs without incurring contention stalls; and non-critical chains are tolerant of the penalties imposed on them in the process of ensuring that critical chains are not delayed. This is an especially important observation because it runs counter to the conventional wisdom: prior work in this area has consistently reported poor IPC figures, leading ultimately to the pervasive view that such losses are inherent in a distributed mode of operation. Second, this result proves that awareness of the critical path, and that judicious allocation of hardware resources to the critical instructions, is an effective means for orchestrating distributed execution. The remaining chapters of Part II are devoted to exploring practical schemes for doing just that.

Practicable potential. In Chapter 7, I use critical path analysis techniques to expose the causes of performance loss in a state-of-the-art clustered machine design. By introducing a new criticality metric, *Likelihood of Criticality* (LoC), I then develop three policies for managing the distributed execution resources.

1. *LoC-based scheduling.* The new LoC metric provides a fine-grained measure of criticality, enabling distinction to be made between two instructions that are often critical, but not as often as one another. A dynamic scheduler equipped to prioritize based on LoC affords the machine a greater ability to give precedence to critical-path instructions, and hence to reduce the effects of resource contention on the instructions that really matter.

2. *Stall over steer.* In code regions in which instruction throughput is limited by long, narrow chains of dataflow, I show that it is better to stall the front-end steering logic when a cluster is full than it is to attempt to continue steering. The LoC metric is

used to guide this selective stalling.

3. *Proactive load-balancing.* On machines with narrow clusters, load-balancing should be performed so as to steer all but the most critical consumer to the cluster holding its producer. Because the most critical consumer does not always appear first in fetch-order, this load-balancing must be proactive: some consumers must be deliberately pushed away from their producer in order to make room for more critical consumers yet to be fetched.

I show that these policies, together, bring performance losses relative to a resource-equivalent monolithic machine to about 6%.

Offline analysis of LoC. Though criticality has previously been viewed as an inherently dynamic property of a program [99], I find that a static criticality profile (*i.e.* a fixed mapping of LoC values onto each static instruction in a program) is just as useful as a fully dynamic scheme for detecting and recording criticality. That is, when the above-mentioned policies are driven by static LoC, their performance is practically the same as that delivered by the online critical path detector and predictor. An important implication of this result is that online critical path predictors are not necessarily tuning into the dynamic behavior of instructions, but merely capturing the average propensity of an instruction to be critical. Viewed in this light, an online infrastructure is superfluous. In Chapter 8, I introduce a novel instruction *trace fabrication* scheme to collect this offline criticality, and I demonstrate the robustness of the ensuing criticality, both across a single program's run, and across runs with different input sets.

Assistance from software. Because instruction criticality tends to be stable, so too do many of the instruction steering and scheduling decisions that are driven by criticality. In Chapter 9, I explore an offline analysis scheme for computing intra-block static instruction steering decisions, and show that this can match, and even surpass, the performance of a

fully dynamic steering scheme. Though hardware is still required to make some dynamic steering and scheduling decisions, the offline technique reduces the bandwidth required of that logic. These results lay the groundwork for what I call a *mostly-static dynamic machine*, a co-designed infrastructure in which the most complex parts (critical path analysis and the bulk of instruction steering decisions) are implemented in an offline software component, but one in which there is still enough dynamic capability in hardware to respond to runtime events that were not — and perhaps cannot be — anticipated statically. I present the results of a preliminary exploration of the opportunities in this regard, and point to a number of potentially fruitful avenues for further research.

1.3 Empirical and analytical methodology

Below, I outline the main empirical tools and analytical techniques I use throughout this dissertation. I do so early on in order to avoid repetition in subsequent chapters and also because some of those techniques have evolved, through the course of my research, into interesting empirical tools in their own right. As such, they warrant explicit mention early on. In fact, a secondary goal of this dissertation is to demonstrate the efficacy of, and hence promote the use of, those experimental tools in a broader context than this work alone.

Pervasive in all my work is the view that instruction criticality has an important role to play in microarchitectural research in general, and in ILDP studies in particular. Indeed, instruction criticality underpins almost all of the results enumerated in Section 1.2. I use it both as an *analytical tool* and as a *mechanism* for improving the performance of distributed execution models. Fields *et al.* use the term *focused policies* to describe the latter, since knowing which instructions are critical affords the designer an opportunity to manage instruction processing judiciously — to focus allocation of resources in favor of those instructions most likely to have an influence on performance [35]. In line with that terminology, I call the use of instruction criticality as an analytical tool *focused analysis*, since

knowledge of the critical path through a program permits the designer to focus attention on just those aspects of behavior that are indeed responsible for the observed performance. Before I introduce these critical path tools, I briefly describe below the simulation infrastructure that I use throughout my work.

1.3.1 Trace-driven simulation

I use a home-grown cycle-accurate simulator to evaluate the various out-of-order microarchitectures I study. It is trace-driven and, beyond assuming a simple load/store Instruction Set Architecture (ISA), is entirely agnostic to the specific ISA from which its traces derive. Though a trace-driven approach does introduce some simplifying assumptions into the simulations, particularly in terms of modeling wrong-path instructions (I stall instruction supply when a branch is mispredicted), the flexibility afforded by the ISA-agnostic approach I have taken has been a boon in this work. In particular, the ability to simulate traces of arbitrary origin facilitated the random trace fabrication work I present in Chapter 8.

I confine my simulations to a small set of machine configurations. I will enumerate specific microarchitectural parameters in the relevant parts of the dissertation, but point out here two high-level assumptions that are common to all machine configurations. First, I model a very aggressive instruction supply. In particular, I assume a perfect instruction cache and unconditional branch predictor, and place no bound on the number of taken branches per cycle. Conditional branches are predicted using an aggressive tournament predictor, the details of which are presented later. Second, I model an aggressive data supply. Though per-cycle issue bandwidth to the memory system is finite (it varies depending on the machine configuration), I do not bound the number of in-flight memory instructions; nor do I model any bank contention in the memory system. Memory disambiguation is also idealized, so loads always issue exactly when their producing stores, if any, have completed address generation. Together, these assumptions render my simulated machines somewhat

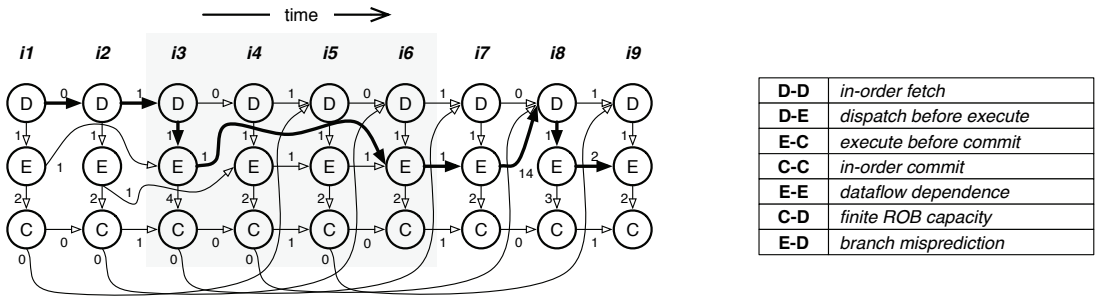
idealized. But I want to stress that my objective in this work is not to model real machines to a high degree of fidelity, but rather to capture just the first-order effects that characterize the various execution models in which I am interested. This is because my focus is on understanding just the basic interactions between program dataflow and microarchitectural constraints, for which purpose I need only model the effects of branch mispredictions, cache misses and finite buffering and issue resources in the core.

In all my empirical work, I simulate portions of the SPEC CPU2000 Integer benchmark programs [94]. These are all compiled to the Alpha ISA using the DEC C compiler (V5.9-005) with peak optimization enabled, but using no profile feedback. I generally run the benchmarks using reduced input sets, and simulate 100-million instructions (after warmup) at each of three checkpointed locations (3-, 5- and 8-billion instructions into the run). In Chapter 8, where I explore the sensitivity of static criticality data to program input, I also simulate the full reference and training input sets, and use a much larger set of checkpoints to ensure better coverage of each run.

1.3.2 Defining instruction criticality

In contrast to the static machines, where the critical path through a program is defined by the length of the longest dataflow chain, dynamic machines, by their nature, render criticality a difficult notion to define. Tune *et al.* developed a scheme that uses various heuristics for detecting instructions likely to contribute to overall performance [98]. For example, instructions that reach the head of the ROB before being ready to commit are very likely to have a direct impact on a program's runtime (*i.e.* they are likely to be on the critical path). Srinivasan *et al.*, in their analysis of the criticality of load instructions, devised an online scheme for determining which loads can be delayed without impacting runtime, and hence which loads are not critical [93].

Figure 1.2 shows a far more sophisticated criticality infrastructure developed by Fields *et al.* [35]. They define criticality in terms of a dependence graph that captures the high-level



D-D	*in-order fetch*
D-E	*dispatch before execute*
E-C	*execute before commit*
C-C	*in-order commit*
E-E	*dataflow dependence*
C-D	*finite ROB capacity*
E-D	*branch misprediction*

Figure 1.2. Critical path model. The dependence graph on the left models a sequence of 9 consecutive instructions in an executing program; the table on the right describes the microarchitectural and dataflow constraints captured by that graph. In this example, the code is executing on a machine with a 4-entry ROB, so C–D edges, which capture the effects of finite window size, connect every 4th instruction. Note also that instruction 7 in the sequence is a mispredicted branch, which induces an E–D edge (from instruction 7 to 8) to model the fact that the correct path instructions cannot be dispatched into the window until the misprediction is resolved. The critical path through this code sequence is highlighted with the thicker dependence graph edges.

behavior of a generic out-of-order machine. Every dynamic instruction is represented by three nodes in that graph, one for each of three steps in its progress down the pipeline: dispatch into the window (D-node), execution (E-node), and retirement from the back-end (C-node). Dependence edges connect nodes within an instruction to reflect the ordering of that instruction's movement through the machine; edges between nodes belonging to different instructions capture the microarchitectural and dataflow constraints imposed on them while they were in flight. For example, each instruction defines a D–E edge to reflect the fact that it must be dispatched into the window before it can be executed. There is a D–D edge between all successive instructions to reflect the in-order fetch constraint; an E–E edge between two instructions models a producer-consumer dataflow dependence. Figure 1.2 enumerates the full set of constraints modeled.

With this model in hand, criticality can be defined precisely. First, the *critical path* is the longest path from the D-node of the first instruction in the program to the C-node of the last. An instruction is called *fetch-critical* if its D-node lies on that path; equally, it is *execute-* or *commit-critical* if its E- or C-node lies on the critical path; instructions that are entirely skipped by the critical path are called *non-critical*. I will often just say an

instruction is *critical* (*i.e.* no qualifier), meaning thereby that it is execute-critical.

I want to stress that these definitions, being derived from a dependence graph that itself reflects an actual execution, constitute an inherently dynamic characterization of criticality: they refer to properties of a single dynamic instance of an instruction, in a specific execution of a program on specific microarchitecture. Criticality, so defined, is *not* a property possessed by a static instruction. All we can talk about in a static context is the propensity of a static instruction's dynamic instances to be critical. Moreover, we can only talk about this in the past tense, the above definitions being applicable only in the context of an instruction trace that has already passed through the machine. They are postmortem in this respect. Thus, to render it a practical tool for managing hardware resources at runtime, Fields *et al.* synthesized a token-based critical path detector from their dependence graph model [35]. This hardware component samples instructions in the pipeline to record various low-level timing events corresponding to each of the dependence graph nodes. Dynamic instructions whose execution (E-node) is thereby identified as critical are used to train a critical path predictor, a PC-indexed table of saturating counters. The predictor is analogous to a counter-based branch predictor, in the sense that it aggregates the past behavior of each static instruction into a binary — in this case, a critical/not critical — decision for subsequent dynamic instances.

1.3.3 Critical path analysis

In addition to serving as the basis for a dynamic critical path infrastructure, the dependence graph model described above is indispensable as a tool in *critical path analysis*. This involves a postmortem analysis of the simulator's execution trace, moving backwards from older to younger instructions, to (conceptually) traverse the dependence graph for the whole execution. At each step, the traversal follows the last-arriving edge coming into the current node, in so doing delineating the critical path for the entire trace; its span is precisely the simulated runtime. Using the path thus identified, it is possible to produce a *critical path*

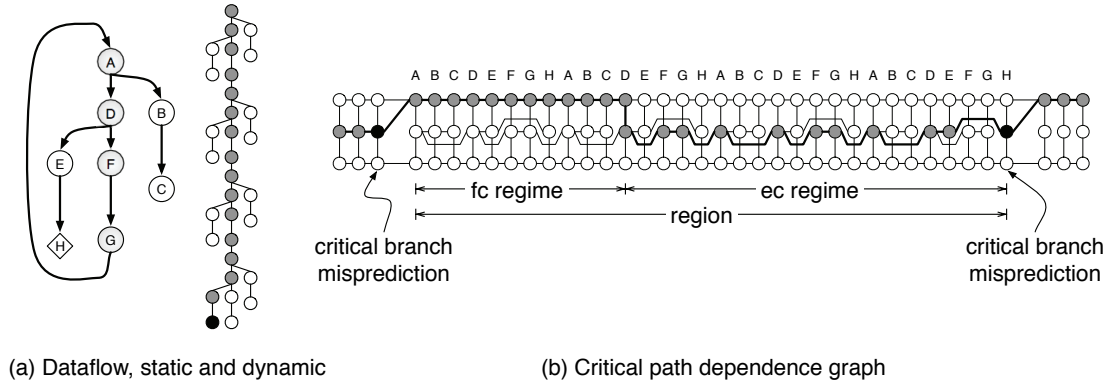
Category		Mnemonic	Description
Front-end	Fetch	*fe*	Time spent waiting for instructions to arrive in the window while the front-end is operating at full speed. Increasing fetch bandwidth would reduce this component of the critical path.
	Fetch stalls	*fs*	Stall cycles arising in the front-end as a result of a full ROB, load/store queue or issue queue. Enlarging those structures would eliminate such stalls.
Execution core	Execution	*ex*	Time spent in instruction execution.
	Resource contention	*rc*	Cycles waited by execute-critical instructions between becoming data-ready and being issued.
	Global penalty	*gp*	Cycles waited by execute-critical instructions for their last-arriving operand to traverse the global communication network.
Memory		*ml*	Cycles spent waiting for data to arrive after an L1 cache miss.
Mispredictions		*mp*	Cycles spent waiting for the pipeline to refill after a branch misprediction. This is a function of both the misprediction rate and penalty.

Table 1.1. Critical path components. The critical path is divided into four main components: time spent in the front-end, the execution core, the memory system, and time spent recovering from branch mispredictions. The front-end category is further sub-divided into time spent on instruction supply itself and time lost at instruction dispatch while waiting for execution core buffer resources to become available. Likewise, the execution core category is divided into sub-categories: time spent on instruction execution proper, and time lost to resource contention and global communication. Monolithic machines will not incur global communication penalties, of course, and they will be largely immune to resource contention stalls.

breakdown — an attribution of cycles to various aspects of behavior. For example, the sum of all E–E edges along the path constitutes the total time spent on instruction execution; the sum of D–D edges represents the net time spent waiting for instructions to get into the window. Table 1.1 enumerates the complete set of categories I make use of.

A critical path breakdown is beneficial in at least three respects.

Diagnosis. Knowing the relative contribution of various factors to overall performance permits accurate diagnosis of the sources of performance loss. For example, knowing that 15% of runtime is spent on the 'gp' category (see Table 1.1) is a guarantee that global



(a) Dataflow, static and dynamic

(b) Critical path dependence graph

Figure 1.3. Execution regimes. The graphs on the left show the static and dynamic dataflow for a small, hypothetical loop. The dependence graph on the right traces the critical path through four successive iterations of that loop, the last of which terminates with a critical branch misprediction. That trace region starts out as fetch-critical (the "fc regime"), where performance is determined by the rate at which instructions are delivered into the window. It then becomes execute-critical (the "ec regime") when execution of instructions on the backward dataflow slice of the next branch misprediction becomes critical.

communication is having a significant impact on performance.[1] By contrast, measuring aggregate global communication — or average global operands per instruction, which is common in many studies — tells one nothing about whether global communication is having an effect. Indeed, an increase in such a metric might even coincide with an improvement performance.

Conceptual model. Understanding the main composition of the critical path provides a conceptual model in which to reason about behavior. First, it lends itself to *interval analysis*, an approach in which a long instruction trace is divided into small, contiguous *regions*, each of which can be examined in isolation from the others [48, 65]. In particular, I make extensive use of a scheme in which critical branch mispredictions are used to delineate each region. At the head of each region is the first correct-path instruction following a branch misprediction; at its tail is the next mispredicted branch, the execution of which is on the program's critical path. Figure 1.3 shows an example of this. Starting at a prior branch misprediction, the dependence graph delineates the critical path through a region compris-

[1] But it does not guarantee that removing those penalties will improve performance by 15%, since their removal might simply expose parallel- or near-critical paths.

ing successive iterations of a small, hypothetical loop; it ends with the misprediction of the loop's terminating branch at the end of the fourth iteration. The region starts out as fetch-critical: the rate at which instructions are delivered into the (probably empty) issue queue is determining performance. It remains fetch-critical until the backward dataflow slice of the next mispredicted branch is fetched into the window, at which point the region shifts into *execute-criticality*. More accurately, the execute-critical regime is entered when the backward slice is delivered into the window at a rate faster than its dataflow height permits issue logic to remove it. The execute-critical regime is therefore characterized by the arrival into the window of instructions whose operands are not yet ready. In the critical path breakdown described in Table 1.1, the span of fetch-critical regions is covered by the sum of the 'fe' and 'fs' components; the execute-critical regions are covered by the rest.

The critical path is not always as simple as this, of course. The relative contribution to overall runtime of fetch- and execute-critical regions depends on a rather complex interaction between a number of factors. In general, low ILP code increases the contribution of execute-criticality by making it difficult for execution to keep up with instruction supply. Fetch-criticality begins to dominate when execution rate matches instruction supply rate, which occurs when ILP is high. Microarchitecture also has a role to play. For example, fetch-criticality arises in the wake of branch mispredictions and long-latency cache misses. Low ILP code can cause a small issue queue to fill up, thereby throttling instruction supply and causing a shift from execute- to fetch-criticality. Such effects are dampened by branch mispredictions, which effectively drain the window and expose their backward slices as execute-critical. Complex though these effects are, the high-level concepts of fetch- and execute-criticality prove to be very powerful tools for reasoning about the first-order effects at work in distributed execution models.

Signature. A critical path breakdown provides immediate, precise and quantified evidence that a machine is *balanced*, in the sense that no one factor is contributing too much,

or too little, to overall performance. Traditionally, IPC figures are used as a means for evaluating a machine's performance, and researchers tend to have a feel for what constitutes a reasonable IPC figure for a given program on a given microarchitecture. However, vagaries in experimental setups can cloud this already vague and informal approach. A critical path breakdown complements an IPC figure by serving as a "signature" of a machine's behavior. This is needed to demonstrate, first, that a given machine does not spend an inordinate amount of time in any one part of the pipeline and, second, to confirm that proposed enhancements do indeed improve those aspects of behavior they claim to target.

1.4 Scope of the dissertation

One of my main goals in this work has been to discover a very general set of principles governing the performance of distributed execution models. To that end, I have aimed to cover a wide spectrum of ILDP designs, and to do so in a manner that focuses only on those interactions between program dataflow and microarchitectural constraints that are likely to pervade a wide spectrum of machine configurations. Nevertheless, the exigencies of experimental work necessarily limit my scope. Below, I enumerate the most important assumptions that are implicit in my experimental choices, and which therefore limit the extent to which I can really claim that my observations are general.

1.4.1 Workloads

As I noted above, I confine my simulations to just the SPEC CINT2000 benchmarks; I do not simulate the floating point (CFP2000) benchmarks. I focus on these specifically because they are representative of a class of program that appears to be inherently resistant to parallelization, and which will therefore benefit most from an architecture that aggressively, and dynamically, pursues ILP. The floating point benchmarks, by comparison, lend themselves to manual and automatic parallelization, and so are liable to benefit more from

architectures that pursue TLP. I want to stress, therefore, that many of the observations I make about properties of program dataflow apply specifically to the control-intensive codes; they do not necessarily generalize to applications in which control flow is highly predictable and in which dataflow tends to be more regular in shape. This does not imply such codes cannot be executed in a distributed way, just that their very different control and dataflow properties preclude a generalization of my results to all SPEC CPU2000 programs.

1.4.2 Static versus dynamic

I take the stance that dynamic machines are inherently better than static machines at finding and exploiting ILP, their implementation problems notwithstanding. I have therefore confined my analysis to dynamic machines, though I acknowledge there is a very large body of work exploring clustered VLIW microarchitectures.

Within the context of dynamic ILDP machines, both static and dynamic instruction steering policies have been proposed. Given the dynamic nature of such machines, it is not surprising that static instruction steering schemes have not been as effective as their dynamic counterparts. My focus in most of the work, therefore, is on ILDP machines with some form of dynamic steering logic in the front-end of the pipeline. Like others, I make the assumption that this steering logic, being predominantly dependence-based in nature, can operate in parallel with register rename logic, and so does not introduce any additional latency in the front-end. In the latter parts of this dissertation, however, I explore the potential of admitting some degree of static decision making in the steering process. But I do not advocate wholly static schemes. Indeed, one of my goals in that work is to try to find the right balance between the inherently dynamic aspects of instruction distribution, and those that are amenable to static implementation.

One of the often-touted benefits of a static approach is that program code can be generated with a target microarchitecture in mind. Though this can introduce an unwanted

coupling between a program's specification and the microarchitecture, I have not explored the potential benefits that a more targeted binary might offer — doing so would necessitate a non-trivial investment of effort in a compiler infrastructure. This issue is perhaps most important in the context of Chapter 4, where I argue that the distribution of ILP in the instruction stream inherently limits the potential of slip-oriented execution. It is conceivable that careful scheduling of code by a compiler targeting a specific slip-oriented machine would mitigate some of those problems. Though I do not believe that a different compiler would change any of my overall conclusions, it should be borne in mind that all of the observations I make about dataflow are, to some extent, dependent on the code generated by the compiler.

1.4.3 Microarchitecture

As I noted earlier, my focus in this work is on the execution core and, more specifically, on the ramifications that a distributed mode of operation has for its ability to exploit ILP. But the execution core is not the only part of the machine that impacts performance. As I mentioned in the preface, an imperfect instruction and data supply fundamentally limit the utility of scaling an out-of-order design. Beyond those perennial problems, there are a number of challenges that fall outside the scope of my work. Among the most pressing are the following.

1. *Memory disambiguation.* Current machines employ an associative load/store queue to dynamically detect aliasing loads and stores. This structure is already a power-hungry and clock-limiting one, and scaling it to support a large-window machine will only exacerbate its problems.

2. *Register file and register rename.* Machines with large windows need a large, multi-ported physical register file to buffer in-flight values. By replicating the register file at each PE, a distributed design can reduce the number of read ports required per struc-

ture, but not the number of write ports (if peak write bandwidth is to be supported). A more challenging problem is posed by register rename, which must be able to handle, say, 8 potentially dependent instructions each cycle. The mostly-static dynamic machine to which I alluded earlier, and to which Chapter 9 is devoted, might offer a solution to this problem.

3. *Interconnect.* I assume in all my experiments that global communication is uniform in latency and unbounded in bandwidth. That is, all values produced in a single cycle can be communicated to all other PEs with a fixed latency. This is reasonable when the PE count is modest, but not so for larger PE counts.

4. *Memory bandwidth.* I assume throughout my empirical work that memory bandwidth can scale with PE count, which is of course not realistic. Prior work in clustered machines has explored both a banked approach, where each PE has direct access to one cache bank [81], and a fully-replicated approach, where the entire first-level cache is replicated for each PE [52]. Neither option is appealing — the former because performance is now impacted by whether locally issued loads end up having to access a remote bank (leading to proposals for cache bank predictors to help guide the steering process); the latter because it is simply an inefficient use of available cache storage.

Finally, a pervasive assumption in this work is that PEs are homogeneous, in the sense that all have the same execution capabilities. Though heterogeneity might make sense in terms of efficient use of silicon resources, it adds another variable to the already hard problem of finding a good distribution of instructions among PEs. I view a solution to the homogeneous case as a necessary first step in solving the more general problem.

Part I

The Laned Machines

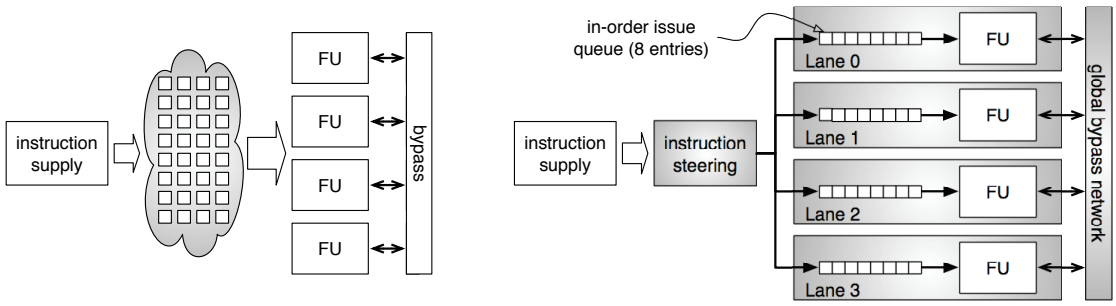
Chapter 2

Introduction

Figure 2.1 shows how a 4-wide monolithic core with a 32-entry issue queue can be partitioned into 4 PEs, each housing an 8-entry *in-order* issue queue. As I noted earlier, I refer to each of the in-order PEs as a *lane* and to the aggregate as a *laned machine*. Instructions are steered by front-end dispatch logic into the tail-ends of the lanes, and issue from those lanes when they reach the head and are data-ready. Although such a machine comprises only in-order parts, it is not an in-order machine. Out-of-order execution is still possible, though now only to the extent that the individual lanes, which operate independently of one another, make progress through the instruction stream at different rates. It is to this *slip-oriented out-of-order execution model* that this first part of the dissertation is devoted.

The laned machines are appealing because of their obvious benefits to design complexity, clock speed and power consumption. They are also conceptually appealing: they offer the prospect of sustaining out-of-order superscalar performance by means of in-order scalar parts. The extent to which that might be possible — the extent to which slip-oriented execution can approach that of dataflow-oriented execution — is the research question I tackle here. I will show that the per-lane in-order constraint introduces fundamental obstacles to achieving good performance, obstacles that are not present when PEs support out-of-order execution. If the goal is to match the performance of a monolithic machine, slip-oriented execution is simply not compelling: it appears (some form of) out-of-order capability is needed locally if effective out-of-order execution is to be achieved globally.

My exploration of slip-oriented execution starts in Chapter 3 with a detailed analysis of the *dependence-based scheduler* introduced by Palacharla *et al.* about a decade



(a) 4-wide monolithic, 32-entry IQ

(b) 4-wide laned machine, 4x8-entry IQs

Figure 2.1. A laned machine. The diagram on the left depicts a 4-wide out-of-order machine with a 32-entry issue queue. The diagram on the right shows how those execution resources can be partitioned into 4 1-wide PEs, each with an 8-entry in-order issue queue. Only the head (oldest) entry in each lane's issue queue is a candidate for issue. Operands are communicated locally without delay, but incur a global communication penalty when they must move to other PEs.

ago [76]. Originally proposed as a complexity-effective alternative to the broadcast-based dynamic scheduling techniques typically used in out-of-order machines, the dependence-based scheduler partitions the issue queue into a number of independent FIFO buffers, permitting only instructions at the heads of those buffers to be eligible for issue each cycle. In this respect, it is exactly analogous to the design depicted in Figure 2.1(b). That is, it implements the slip-oriented execution model. But the design partitions only the issue queue, not the functional units, so it does not introduce a global communication penalty into the execution model. It represents, therefore, a best-case scenario for the performance potential of slip-oriented execution and, as such, is an ideal starting point for my exploration. The original evaluation of the dependence-based scheduler reported very good performance results relative to a monolithic superscalar, and so paints a very promising picture for the prospects of slip-oriented execution. However, I will show in Chapter 3 that the dependence-based scheduler is liable to perform much worse than the prior work would suggest. Furthermore, I will show that its problems are innate, deriving from interactions between basic properties of dataflow and its particular dependence-based instruction steering policy. The original study by Palacharla *et al.* is therefore misleading, in the sense that its performance results do not actually say anything about the dependence-based scheduler; the good re-

sults arose, not because the scheduler is effective, but because constraints outside of the execution core hid its performance problems.

These negative results do not imply that slip-oriented execution itself is not viable, just that the scheduler — specifically, its dependence-based steering policy — is not very good at exploiting ILP. The question as to the true potential of slip oriented execution therefore remains open. I turn to that problem in Chapter 4. Rather than embark on an exploration of a new suite of heuristic-based steering policies, I step back from design specifics and explore the underlying factors affecting the performance of any machine implementing the slip-oriented model. I show that these machines can, in principle, match monolithic performance, but that achieving their potential is not practicable. Specifically, I show that matching the performance of modern dynamically-scheduled designs demands that a laned machine be able to simultaneously manage a large number of active dataflow chains, many more than the amount of ILP typically extracted from the code. Because each lane imposes an in-order constraint on execution, this requirement, in turn, demands either that the active dataflow chains be carefully interleaved among the available issue queues, or that enough lanes be provided for each to be buffered at its own issue queue. Using an abstract model for reasoning about the performance of these machines, I show that the former option is fundamentally hard, in the sense that it necessitates instruction steering hardware that would be too complex to build. The latter option would demand so many lanes that the machine would be overwhelmed by overheads like global communication.

The conclusions to be drawn from this analysis are not very positive. In short, if a partitioned design is to match the IPC performance of the dataflow-oriented execution model, then some form of out-of-order execution must be supported at each of the PEs; in-order execution is not sufficient. Though there are, of course, compelling benefits to in-order designs, we are already building fast monolithic machines of modest superscalar dimensions and manageable complexity. A wide-issue, ILP-inefficient laned machine therefore makes little sense.

These results also have important ramifications for CMPs. The analysis I present in Chapter 4, being built upon a very general notion of what constitutes a laned machine, applies equally to *horizontal fusion* of in-order cores. Such designs would be extremely compelling in a multicore environment because they would facilitate the design of simple in-order substrate, which is ideal for TLP-rich workloads, and at the same time would cater to single-thread workloads by dynamically fusing the in-order cores into a single wide-issue, out-of-order machine. Though additional overheads are introduced in this fusing process, an aggregate machine comprising in-order execution units is an instance of the slip-oriented execution model and, as such, will suffer from the same performance limitations I expose in Chapter 4. It seems likely, therefore, that some form of out-of-order execution will have to be supported in future CMPs if single-thread performance is to remain at the levels we have come to expect from modern general-purpose processors.

Chapter 3

Dependence-based scheduling revisited[†]

The *dependence-based scheduler* was proposed as a complexity-effective alternative to conventional, broadcast-based dynamic scheduler designs. It partitions the issue queue into a number of FIFO buffers, permitting only those instructions at the heads of each to be eligible for issue each cycle. Scheduling logic benefits from such an organization because it need monitor the readiness of, and select from among, only the heads of each FIFO buffer; complex broadcast-based circuitry is entirely eliminated.

In their 1997 study, Palacharla *et al.* showed that this scheduler is capable of delivering IPC which rivals that of a monolithic machine, with average loss being about 5% [76]. Though that study was conducted over a decade ago, its good results are important in the context of this dissertation because they imply that the slip-oriented execution model is not a very restrictive one, and hence that a machine comprising only in-order PEs is a promising direction for ILDP designs. Moreover, the good performance results embed a rather fundamental statement about dynamic dataflow and, as such, have very broad implications for computer architecture in general. This is because the dependence-based scheduler exploits properties of dataflow to overcome the restrictions imposed by its execution model. Specifically, it uses dataflow dependences to control the allocation (steering) of instructions to the FIFO buffers, slotting consumers directly behind the producers of their operands and thereby ensuring that instructions are always dataflow dependent on those ahead of them in their respective FIFO (hence the scheduler's name). This simple invariant ensures that the in-order issue constraint is always *subsumed* by dataflow constraints. Such a steering

[†] The content of this chapter derives from work published at the 6th Annual Workshop on Duplicating, Deconstructing and Debunking [85].

policy will only be effective to the extent that the instruction stream embeds dataflow that lends itself to being mapped onto the FIFO buffers in this manner. That good IPC has been demonstrated means, therefore, that dataflow is indeed amenable to being decomposed into a few chains of dependent instructions, the independent execution of which suffices for extracting the available ILP.

My own evaluation of the dependence-based scheduler does not corroborate such conclusions. In fact, I find that dependence-based scheduling incurs losses about sixfold higher than those published in the original study. Taken by themselves, my results paint a very different picture about dataflow and about the efficacy of the slip-oriented execution model. In this chapter, I present a detailed analysis to uncover the sources of discrepancies between my results and those originally published. I begin in Section 3.1 with a brief review of the dependence-based microarchitecture, together with a summary of the performance results originally published by Palacharla *et al.* I also describe there my own evaluation of the microarchitecture. Its very different results serve as the motivation for the detailed analysis presented in the remainder of this chapter.

In Section 3.2, I explore the basic factors at work in a dependence-based machine. Critical path analysis reveals that the machine suffers from a preponderance of *fetch-criticality*, meaning it is operating in a regime in which performance is determined mainly by the rate at which instructions are being delivered into the window. This is caused by the machine's dependence-based steering rules, which frequently stall instruction dispatch for lack of a suitable slot into which the next instruction can be placed. This problem arises, ultimately, because of basic properties of dynamic dataflow. Simply put, dataflow does not readily decompose into just a few dependence chains, but rather into a large number of chains, most of them very short. The net effect is "shallow and wide" distribution of instructions across the FIFO buffers, followed by dispatch stalls until one of the buffers drains. The issue queue's effective capacity is thus dramatically reduced, and so is lost of one the main benefits of a conventional issue queue — that of *buffering*, or *shelving* [89], of instructions

in order to *decouple* dispatch from the order and rate at which instructions can be executed. Without this capability, the machine is restricted in terms of its ability to exploit ILP found across loop iterations, its ability to make forward progress through the instruction stream to reach important instructions that lie ahead, and its ability to find and exploit memory-level parallelism (MLP).

With a proper understanding of the machine's behavior in hand, I then offer explanations for the good performance results reported in the original study by Palacharla *et al.* In Section 3.3, I postulate that the baseline machine into which the dependence-based scheduler was introduced, and against which it was evaluated, was seldom stressing the execution core. Its critical path was most likely dominated by fetch-criticality and by recovery from branch mispredictions. In short, the new scheduler introduced its fetch-criticality problem *in the shadow of* a pre-existing instruction supply constraint. Its deleterious effect on performance was thus hidden. In my own evaluation of the machine, the underlying instruction supply constraint is less severe, so the scheduler's problems come to the fore. This does not imply the original study's choice of baseline microarchitectural parameters were wrong, but it does mean the efficacy of the dependence-based scheduler was never really tested.

3.1 Background

Modern superscalar machines support the dataflow-oriented out-of-order execution model by means of a complex, broadcast-based scheduling technique. This involves two mutually dependent operations. In the first, called *select*, instructions whose operands are ready, and for which execute resources are available, are issued to the functional units. The second, called *wakeup*, involves notifying all instructions in the issue queue of the imminent availability of values to be produced by the selected instructions. Consumers of those values thus become eligible for selection in subsequent cycles. In their 1997 study, Palacharla *et*

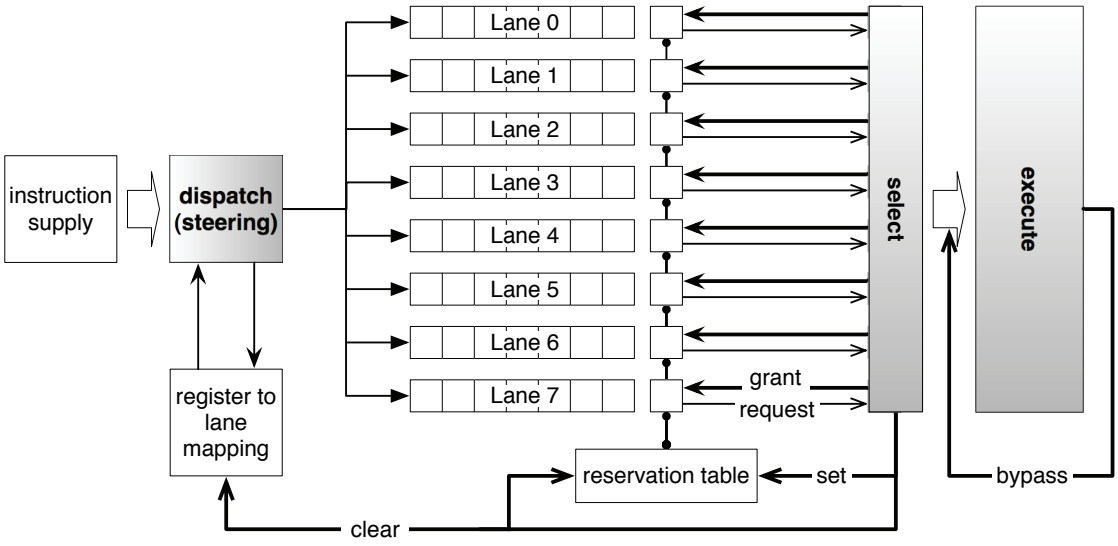


Figure 3.1. Dependence-based scheduling. (*cf.* Figure 2.1(b).) The diagram shows a partitioning of a 64-entry issue queue into 8 lanes (FIFO buffers), each 8-deep. The head (oldest) entry in each lane is conceptually distinct from its successors since only it is a candidate for selection. Operand availability is communicated to all head entries via the reservation table, whose 1-bit entries are set when the producer of the corresponding value is selected, and cleared when that value becomes available. Steering logic records the lane to which the producer of each architected value is sent; scheduling logic clears this record (if it is still current) when that producer is issued.

al. found that the scheduler's latency — wakeup delay, in particular — dominates that of other microarchitectural components. And since the need for back-to-back issue of dependent instructions precludes pipelining its operation, the scheduler imposes a lower bound on the machine's achievable clock period.

3.1.1 Dependence-based scheduling

The dependence-based scheduler was proposed as a means for facilitating a faster clock. Figure 3.1 shows its design. The issue queue is partitioned into a number of FIFO buffers (I will call them *lanes*), each of which holds a subset of all dispatched instructions.[1] Dispatch logic *steers* instructions into the tail-end of the various lanes, using dataflow dependences among them to make its decisions. The heuristic used for this purpose proceeds as follows.

[1]This is a slight abuse of my own terminology. I have up to this point been using the term *lane* to refer to an in-order PE, not to just an in-order issue queue. The analogy is hopefully clear, nevertheless.

1. Pick a non-full lane whose tail entry holds the producer of one of the dispatching instruction's operands.
2. If no such lane can be found, pick an empty lane.
3. If no empty lane is available, stall.

To paraphrase: an instruction is slotted into a non-empty lane if, and only if, (one of) its producer(s) immediately precedes it there. This will not be possible when all producers have issued or none of them reside at the tail of a non-full lane. In such cases, an empty lane is sought. If that also fails, dispatch is stalled.

Because all instructions within a lane are dependent on those ahead of them, only the heads of each lane are candidates for issue. The select logic therefore need only concern itself with a small subset of the issue queue (8 out of 64 entries in Figure 3.1). For the same reason, back-to-back issue can be achieved by advertising result availability to only the heads of each lane. These simplifications have enormous implementation benefits. Most importantly, elimination of the long wires and all the comparators needed by the broadcast-based wakeup logic reduces scheduler latency enough to remove it from the processor's timing critical path. The same simplification offers savings in terms of area and power. Of course, all these benefits must be weighed against the machine's IPC performance potential.

3.1.2 Performance

The dependence-based scheduler exploits properties of dataflow to overcome the constraints imposed by its restricted execution model. Because the steering logic permits instructions to collocate at a lane only if they are part of a dataflow dependence chain, it is guaranteed that the in-order constraint imposed by that lane is always subsumed by dataflow constraints. In other words, dataflow dependences render the in-order constraint benign — but only so long as a program's instruction trace embeds dataflow that is amenable to being mapped onto the lanes by means of the steering rules described above. It is not immediately obvious to what extent this ought to be possible. It depends on the

Instruction supply	perfect instruction cache. perfect unconditional branch prediction; modest gshare conditional branch predictor.
Front-end	8-wide, unknown depth.
Window	128-entry ROB. 64-entry unified issue queue.
Execute	8 universal units, 4 memory ports. all instructions unit latency. loads wait for older store addresses.
Memory	L1: 32KB, 2-way, 1-cycle hit. L2: perfect, 6-cycle hit.
Back-end	16-wide.

Table 3.1. Baseline machine parameters used by Palacharla *et al.*

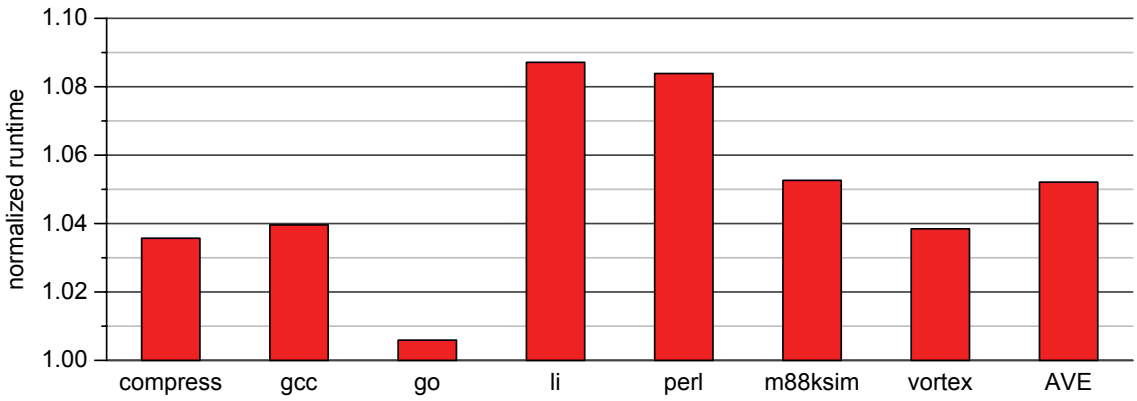


Figure 3.2. Performance results obtained by Palacharla *et al.* The graph plots the runtime of the dependence-based machine relative to that of the monolithic machine described in Table 3.1. Data is interpolated from the absolute IPC numbers published by Palacharla *et al.* The simulated programs form part of the SPEC CPU95 benchmark suite.

shape of, and distribution of ILP within, a program's dynamic dataflow graph, neither of which is an easily characterized property.

Palacharla *et al.* render the whole question moot with an empirical evaluation of their scheme. Figure 3.2 reproduces their results. An 8-wide baseline machine (see Table 3.1) is compared to a dependence-based counterpart that partitions the 64-entry issue queue into 8 lanes, each 8-deep. Instruction dispatch is modified to steer instructions into the lanes using the heuristic described above. As Figure 3.2 shows, the dependence-based machine experiences very modest slowdowns: about 5% on average, never worse than 10%.

These results are significant because they confirm that all of the aforementioned imple-

Instruction supply	perfect instruction cache. perfect unconditional branch prediction; aggressive tournament conditional branch predictor.
Front-end	8-wide, 10 stages to dispatch.
Window	128-entry ROB. 64-entry unified issue queue.
Execution	8 universal units, 4 memory ports. latencies similar to Alpha 21264. ideal memory disambiguation.
Memory	L1: 32KB, 2-way, 2-cycle hit. L2: perfect, 12-cycle hit.
Back-end	8-wide.

Table 3.2. Baseline machine parameters used in my evaluation. (*cf.* Table 3.1.)

mentation benefits can be won at modest cost to IPC. More importantly, the ability to sustain good IPC *in spite of* a restricted out-of-order execution model has two important implications. First, the flexibility afforded by the less constrained dataflow-oriented out-of-order execution model appears to be superfluous; it is possible to exploit dataflow properties to build a simple, complexity-effective scheduler. Second, dataflow has the propitious quality that it can be quite readily decomposed into (just a few) chains of dependent instructions, the parallel and independent execution of which suffices for extracting ILP. The latter is an especially important result because it has broad relevance, particularly for the numerous studies that propose buffering instructions as per their dataflow relationships [28,52,53,64].

These compelling results stand in stark contrast to those I obtained. I incorporated a version of the dependence-based scheduler into my own simulator infrastructure so as to explore the slip-oriented execution model in more detail. Given the broader context of my analysis, it was *not* my aim to replicate the original numbers. I therefore evaluated a different benchmark suite (SPEC 2000 instead of SPEC 95) and my simulator parameters differed somewhat from those used in the 1997 study. However, I share with Palacharla *et al.* the overall objective of modeling, not a specific state-of-the-art microarchitecture, but one in which the execution core is exercised to the fullest. As Table 3.2 shows, I assumed an aggressive instruction supply, with ideal instruction cache and perfect prediction of unconditional control flow. But, like Palacharla *et al.*, I retained imperfect conditional branch

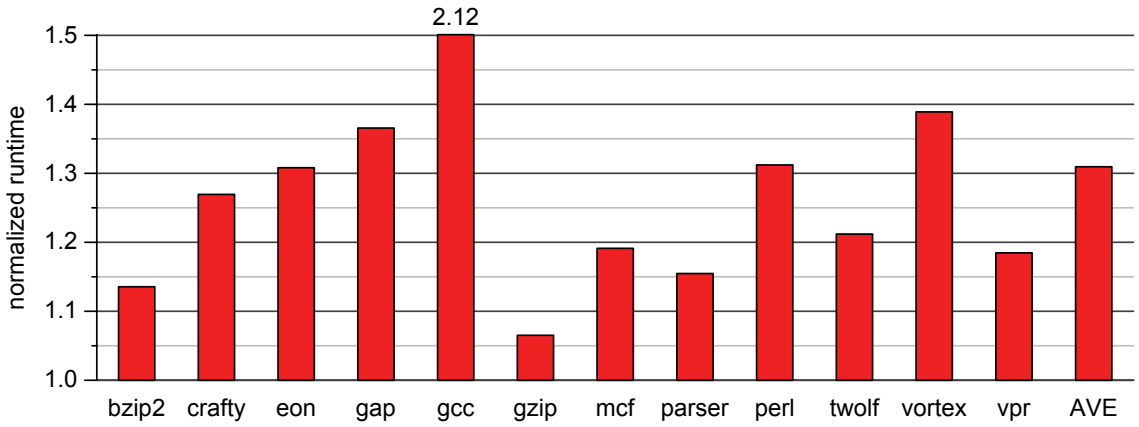


Figure 3.3. My evaluation of dependence-based scheduling. Like Figure 3.2, the graph plots the runtime of an 8-laned dependence-based machine relative to a machine with a 64-entry monolithic issue queue. Note how the scale on the y-axis now differs from that in Figure 3.2.

prediction because the effect of mispredictions can be important for scheduling. Likewise, I modeled an aggressive, but imperfect, data-supply to capture the effects of variable latency operations.

My simulation results appear in Figure 3.3. Instead of modest penalties in the region of 5%, I find slowdowns are, on average, sixfold higher at about 30% and, in the majority of cases, are at or above 20%. The case of `gcc`, which is a clear outlier at more than double the runtime, also hints at a lack of performance robustness in the dependence-based scheme. To put these performance losses in perspective, it is worth noting that they are not very different from what my monolithic baseline machine achieves when its issue queue is reduced to 25% capacity (16 entries). Equally, a scheduler that pipelines its wakeup-select logic has been shown to suffer very similar slowdowns [95].

Although these newer results might render the dependence-based scheme less appealing from a complexity-effective point of view, it is perhaps tempting to dismiss them as an artifact of differences between the respective experimental frameworks. Of course, it is ultimately those differences that are to blame, but such a coarse diagnosis is unsatisfactory, for two reasons. First, it does not explain why trends in *relative* performance are so different; one would expect simulator vagaries to be largely accounted for by the comparative

evaluations. Second, it does not answer the more fundamental question about the potential for exploiting basic properties of dataflow to build a scheduler based on the independent execution of dependence chains.

3.2 Deconstructing dependence-based scheduling

In this section, I analyze the behavior of (my implementation of) the dependence-based machine in detail. I show, first, that it suffers from an instruction supply problem and, second, that it is an interaction between the steering rules and basic properties of dynamic dataflow that are to blame. Underpinning both of these results is *critical path analysis*, the technique I described in Chapter 1 (Section 1.3.3). This permits me to *quantitatively* characterize the behavior of the dependence-based machine. Although aggregate statistics might well be used to arrive at some of the conclusions I present below, a critical path analysis is the most direct and accurate means for doing so. It permits me to zoom directly in on those aspects of behavior that are truly having an effect on performance. In this regard, aggregate statistics can be very misleading because they do not always correlate with observed IPC.

3.2.1 Where the cycles went

Recall from Chapter 1 (Section 1.3.2) that I use the dependence graph model developed by Fields *et al.* to delineate the critical path through an executing program [35]. This involves a postmortem analysis of the simulator's execution trace, moving backwards from younger to older instructions to identify the microarchitectural and dataflow constraints that, together, determine the observed runtime. The critical path, thus identified, permits attributing runtime cycles to those aspects of behavior in which I am interested. Figure 3.4 shows the resulting critical path breakdown for the baseline and dependence-based machines. A clear difference between the two immediately stands out. In the dependence-based machine,

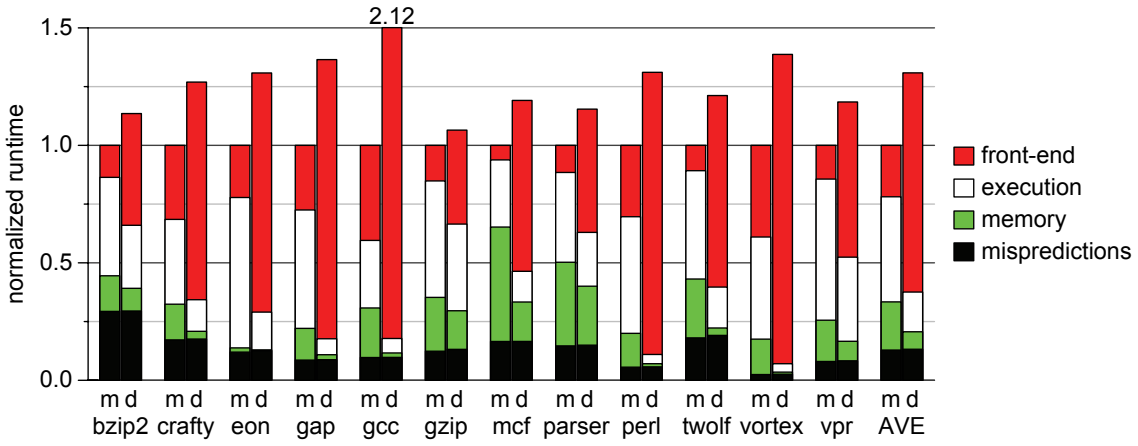


Figure 3.4. Critical path breakdowns. Each pair of bars shows the critical path breakdown for the monolithic baseline ('m') and the dependence-based ('d') machines. Cycle counts are normalized to the former. The categories for the critical path breakdown are described in Table 1.1 (page 32).

about 65% of runtime is spent in the machine's front-end, meaning performance is determined mainly by the rate at which instructions are delivered into the window; time spent on instruction execution and in the memory system accounts for only 25% of runtime. The monolithic baseline machine is quite the opposite: about 25% of time is spent waiting for instructions, and 65% is spent on instruction execution and cache misses. Thus, performance of the dependence-based machine is determined primarily by the rate at which instructions are delivered into the window; that of the monolithic baseline by the rate at which they can be issued from it. More succinctly, the former operates in a predominantly *fetch-critical* mode; the latter, in an *execute-critical* one.

A fetch-critical regime is indicative either of insufficient front-end bandwidth or of back pressure in the pipeline preventing instructions from dispatching into the window. The former is certainly not the problem given that both the monolithic and dependence-based machines share the same aggressive front-end architecture. That leaves, then, the back-pressure problem as the culprit. This usually arises when a resource constraint in the execution core — a full issue queue, for example — prevents instructions from being dispatched. However, as the data in Figure 3.5 shows, ROB and issue queue occupancy in the dependence-based machine are both at 30% for more than 75% of the time; both

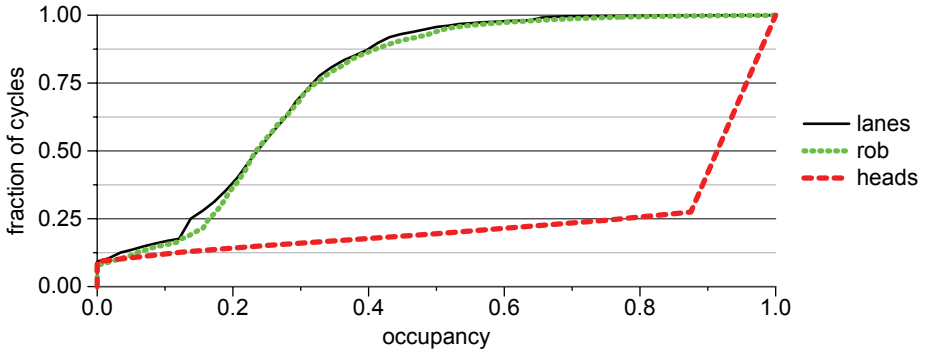


Figure 3.5. Execution core occupancies. The graph shows a cumulative distribution of the cycle-by-cycle reorder buffer occupancy ('rob') and aggregate issue queue occupancy ('lanes'). The 'heads' plot shows the occupancy of the 8 slots at the front of the lanes (*i.e.* one slot from each lane). Data is averaged across all the SPEC CINT2000 benchmarks.

structures average about 25% occupancy.[2] Clearly, a window buffering constraint is not the problem.

Figure 3.5 also shows that occupancy of the lane *heads* is almost always 100%. That is, although aggregate lane occupancy is low, all lanes tend to have at least one instruction in them at any one time. It appears, therefore, that instead of slotting behind one another within the lanes, instructions are being sprayed across them. To confirm that this is indeed happening, I profiled the types of steering decisions being made by the dispatch logic. I classify steering decisions into one of three categories: *collocations*, *spills* or *free-steers*. Collocation occurs when a consumer is successfully slotted behind one of its producers. An instruction is spilled (to an empty lane) when collocation is not possible, either because the producer to be followed is not the last instruction in its lane (something else got there first), or because that producer is at the tail end of a full lane. Free-steers (to an empty lane) happen when all producers have already been issued, so there is no instruction toward which the consumer can be steered. Figure 3.6 shows the relative frequency at which these different types of decisions are made. The data confirms that the dominant behavior for steering is to send instructions into an empty lane: about 55% of all instructions are

[2]This data is consistent with observations made by Michaud and Seznec [64], who found that a dependence-based window with n FIFO buffers is equivalent (from an IPC performance point of view) to a monolithic window with $2n$ entries. At 25% occupancy, the 8-laned issue queue is averaging about 16 instructions in-flight at any one time.

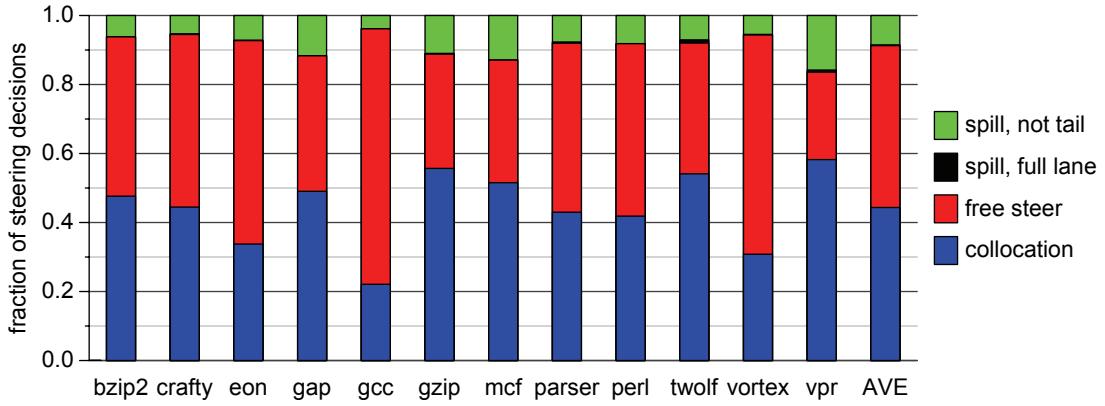


Figure 3.6. Steering decision breakdown. Each bar shows the relative frequency at which collocations, free-steers and spills occur. Collocations occur when a consumer successfully slots directly behind its producer in one of the lanes. Free steers occur when there is no in-flight producer toward which a consumer can be sent. The spill category captures instructions that had a producer in flight, but could not be steered toward it, either because the producer was at the tail of a full lane ('spill, full lane'), or because there was an earlier consumer that managed to collocate with that producer ('spill, not tail').

either spilled or free-steered. In addition, spills arise almost exclusively because an earlier consumer managed to collocate (the 'spill, not tail' category), not because lanes are filling up. Clearly, making lanes deeper is not going to solve any problems. Nor is it a means for scaling aggregate issue window size.

The foregoing points to a scenario in which instructions are being spread "shallow and wide" across the lanes rather than being stacked up behind one another within the lanes. Recalling the dependence-based steering rules described in Section 3.1.1, such behavior would lead inevitably to frequent *dispatch stalls*. That is, if dataflow is tending to spread outward, then it will frequently be the case that the next instruction to be dispatched will not depend on any of the instructions at the tails of the lanes; the steering rules stall under such circumstances. The prevalence of free-steers is then also accounted for because, by the time a lane is emptied and dispatch resumes, the instruction that was stalled will no longer have any producers in flight (*i.e.* it will be free-steered to the empty lane).

Before I zoom in on the principal factors that underlie this shallow and wide distribution of instructions, I want to briefly step back to reflect on the role played by critical path analysis in the above diagnoses. It might well be claimed that I could have made simi-

lar observations simply by recording, for example, aggregate data on front-end dispatch throughput in the two machines: this would surely expose the instruction supply problem just the same. Indeed it would, but relying solely on aggregate data runs the risk of paying too much attention to factors that have little or no bearing on performance. A critical path analysis, however, provides a direct and immediate measure of which factors influence performance and, more importantly, of their relative contributions to the observed runtime. That fetch criticality so dominates the critical path signature of the dependence-based machine constitutes unequivocal evidence of an instruction supply problem. It is this evidence which *justifies* zooming in on aggregate data on window occupancy and steering decisions because it is now known that it is precisely those factors that determine the machine's performance. A more subtle benefit of critical path analysis will come to the fore in the next section, where I explore the main causes for steering-induced dispatch stalls. In this case, it is the conceptual model upon which the critical path infrastructure is built that provides the insights necessary to diagnose and understand those causes. Moreover, it provides the means necessary for understanding why different benchmark programs vary so much in terms of the performance impact of dependence-based scheduling.

3.2.2 Accounting for dispatch stalls

The steering behavior described above might be caused by a number of factors. For example, the shape and distribution of ILP in the instruction stream are clearly both important. But other factors such as the dynamic distance between producers and their consumers, branch mispredictions and cache misses all have a part to play. Nevertheless, it is properties of dataflow that have the first-order effect. Simply put, the problem is that dataflow does not decompose into just a few dependence chains. Rather, it tends to comprise a large number of chains, most of them very short. I now show that this can be accounted for by the manner in which small loops dynamically unwind in the window. Below, I describe this process in detail, together with its principal repercussions for performance.

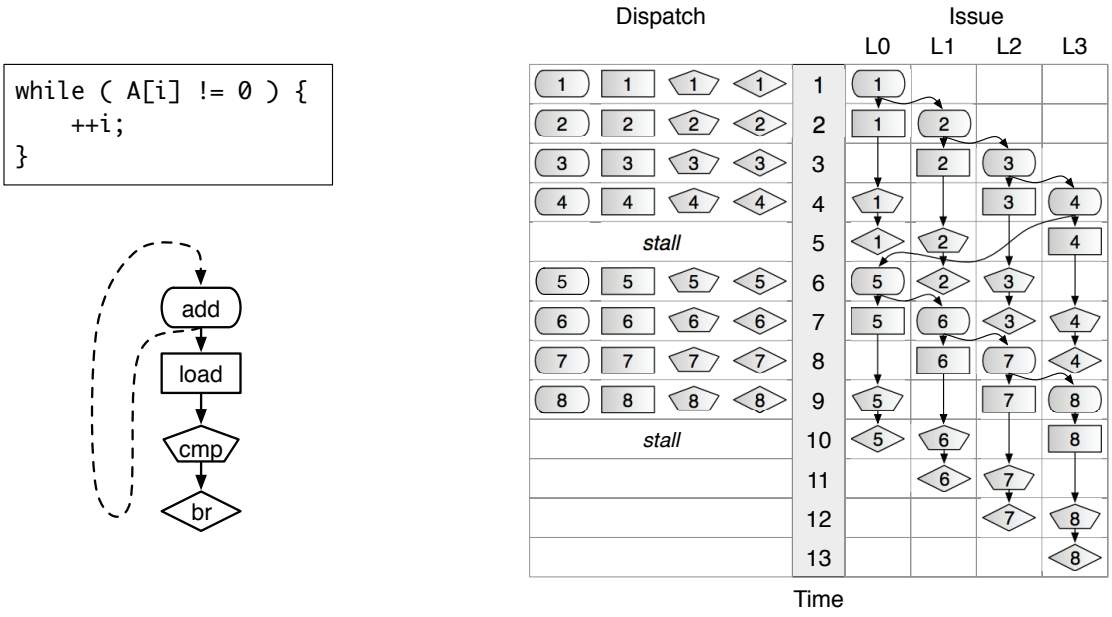


Figure 3.7. Stalls throttle loop initiation. The diagram shows how a hypothetical loop would execute on a dependence-based machine. The loop's source code and corresponding static dataflow appear to the left of a timing diagram showing dispatch and issue of 8 successive iterations (instructions are annotated with the iteration to which they belong). The timing diagram models the operation of a machine with 4-wide fetch and execute bandwidth and 4 lanes in the execution core; loads all hit in the cache and have a 2 cycle load-to-use latency. Dispatch stalls arise every 5 cycles because lanes are consumed by dispatch faster than they are cleared by issue.

Converting execute- into fetch-criticality

Figure 3.7 illustrates a typical "spine-and-ribs" dataflow structure induced by the dynamic unrolling of a small loop. The spine consists of induction variable computations (the add instruction), and the ribs the intra-loop work that branches off the spine (the load-compare-branch sequence). Although it is somewhat contrived, the example distills a basic process at work in the dependence-based machine: as a loop is fetched into the window, it decomposes into small dependence chains (the ribs), causing successive iterations to spread *across* lanes. This behavior arises because dataflow *diverges* — and so spawns a new dependence chain — at each induction variable update. And because loop iterations are fetched faster than they can be executed, this rapid consumption of lanes incurs one dispatch stall every 5 cycles. Those stalls translate into a sustained IPC of 3.2 for the hypothetical code sequence, a 20% loss relative to the peak IPC of 4 that a monolithic machine would achieve. This

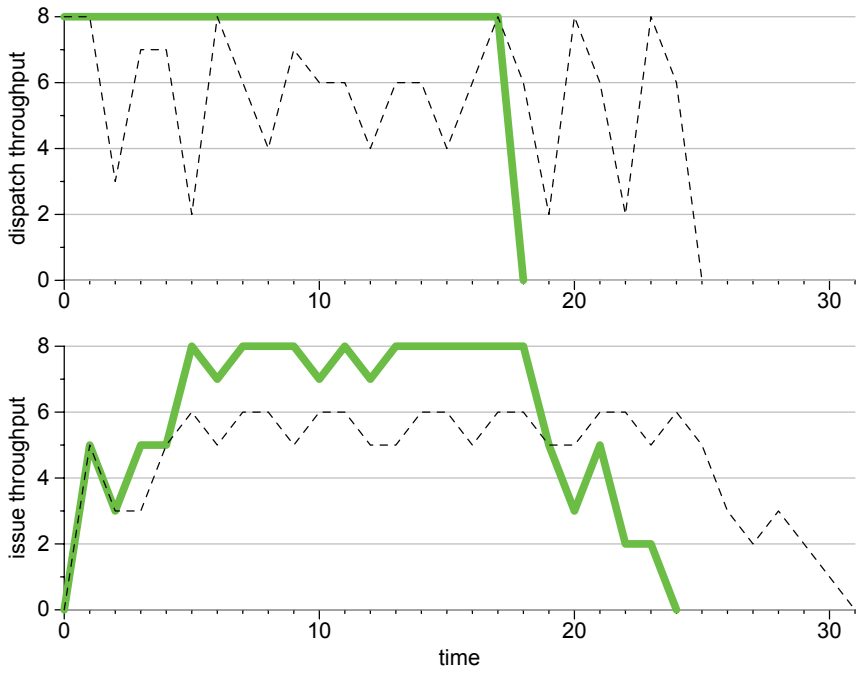


Figure 3.8. Throttling loops in the `twolf` benchmark. The two graphs show dispatch-induced stalls at work in real code. Each shows cycle-by-cycle dispatch (top) and issue (bottom) throughput for a short trace from 9 successive iterations of the inner loop in function `dbox_pos_2` in the `twolf` benchmark. The thicker, pale line tracks dispatch and issue activity on a machine with a monolithic window; the thinner, dashed line tracks that on a dependence-based machine.

problem arises simply because the dependence-based machine cannot deliver instructions into the window fast enough to expose all the available ILP. *It converts execute-criticality into fetch-criticality.*

I want to stress that the stalls incurred by the dependence-based machine in the above example cannot be blamed exclusively on the slip-oriented execution model. For example, if the steering logic would ignore dataflow dependences, opting instead to allocate a separate lane to each static instruction, then the machine would be able to sustain its peak execution rate. In this particular example, therefore, it is the requirement that instructions be steered as per dataflow dependences that is to blame for the performance loss; it is not an inherent problem introduced by the restricted execution model. Whether this fact can be generalized to less contrived pieces of code is a subject I return to in the next chapter.

Figure 3.8 shows conversion of execute- into fetch-criticality at work in real code. The graph contrasts dispatch and issue activity over successive iterations of a hot loop in the

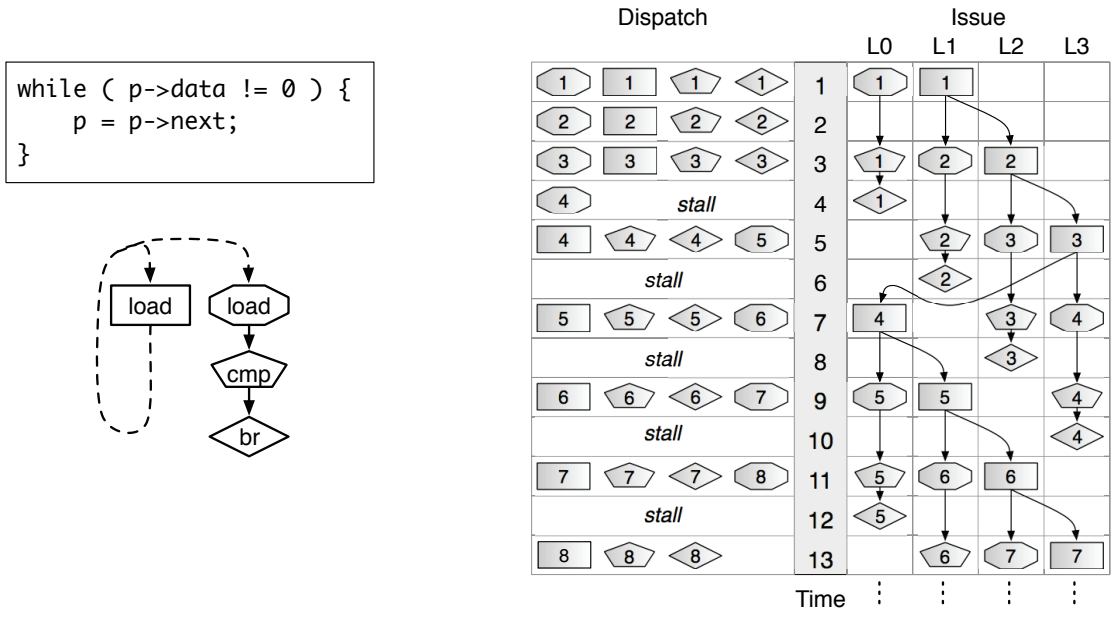


Figure 3.9. Front-end stalls can be benign. Like Figure 3.7, the timing diagram shows a hypothetical code example executing on a 4-wide dependence-based machine. In this case, the throttling effect occurs in the shadow of latencies imposed by low-ILP code.

`twolf` benchmark. That loop has a very high initiation rate and enough ILP to keep the machine operating at close to peak performance. The top graph shows that instruction supply rate in the dependence-based machine (dashed line) is being throttled relative to that in the monolithic machine (solid line), with the latter delivering the trace into the window much quicker than the former. This difference in instruction throughput prevents the available ILP from being exposed to the dependence-based machine's issue logic, causing it to take longer than the monolithic machine to execute the trace (bottom graph).

Although this effect is certainly the principal cause for most of the IPC losses, it is not, in general, a sufficient condition for them. As a counterpoint to the previous example, the hypothetical loop in Figure 3.9 shows that peak performance can be achieved even in the presence of many stalls. In this case, the loop's induction variable updates have longer latency, so the peak initiation rate and, hence, the available ILP, is now lower. Thus, while lanes are consumed very rapidly by this loop, the ensuing stalls do not manifest themselves as performance loss; they are not exposed on the critical path. Figure 3.10 shows that real

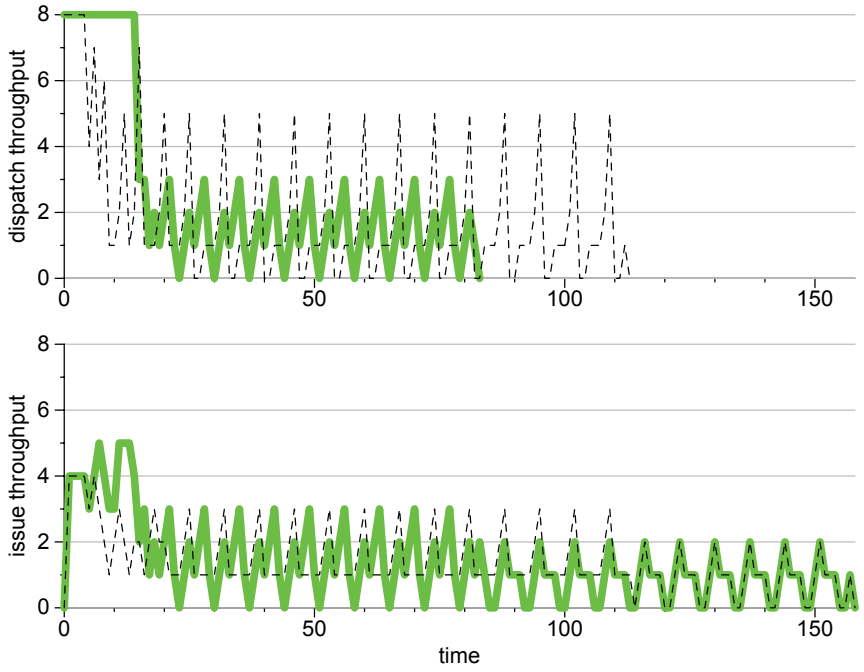


Figure 3.10. Benign stalls in the `gzip` benchmark. As per Figure 3.8, the two graphs show dispatch (top) and issue (bottom) in a short instruction trace, in this case one comprising 22 iterations of the main loop in the `updcrc` function in the `gzip` benchmark. The thicker, pale line tracks dispatch and issue activity on a machine with a monolithic window, and the thinner, dashed line tracks that on the dependence-based machine.

code exhibits exactly this kind of behavior. In this case, the trace is from a hot loop in the `gzip` benchmark, where available ILP is very low. Although dispatch stalls are clearly throttling instruction supply relative to the monolithic machine (top graph), the lack of available ILP means the throttling effect is benign: both the monolithic and the dependence-based machine finish executing the trace at exactly the same time (bottom graph). The fact that this loop is so hot in `gzip` explains why the dependence-based machine performs so well on this benchmark (recall Figure 3.3) — in spite of a very large number of steering-induced stalls.

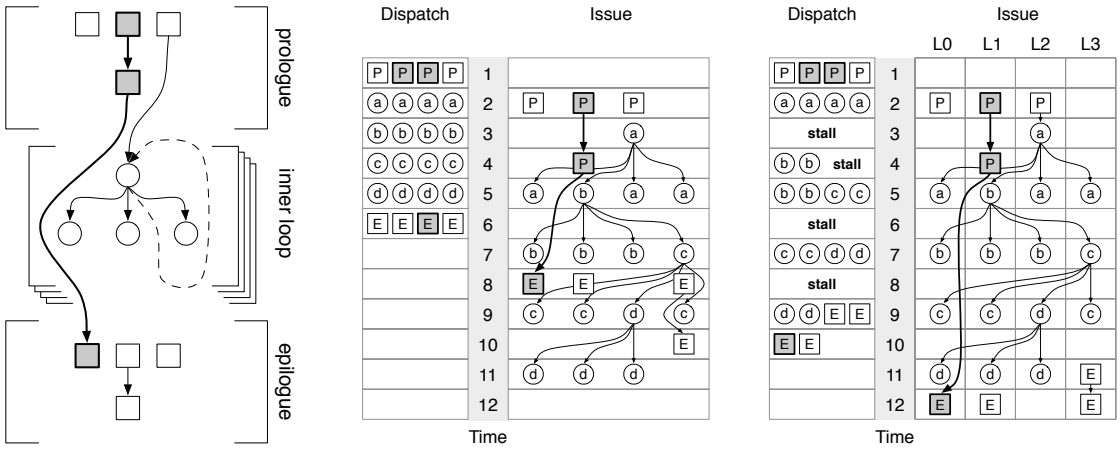
In general, the extent to which execute-criticality is converted to fetch-criticality is determined by the interaction of two processes. The first is the rate at which instructions can be delivered into the window, which is determined by how quickly lanes are consumed by each iteration of the loop and by how long each iteration takes to complete its execution. Loops with more internal ILP and with longer latency operations within each iteration will

consume lanes quickly and will block each lane for longer. The second process is the peak ILP that is available from dynamically unrolling the loop, and how many times the unrolling must occur in order to expose that ILP. When loop initiation rates are low, fewer iterations will have to be unrolled to expose all the available ILP, which mitigates the instruction supply problem.

The foregoing implies that adding more lanes to the dependence-based machine would alleviate its problems. This is a subject I explore in detail in Chapter 4, but I will state here that, indeed, performance does steadily improve as more lanes are added. But it is not until about 24 lanes are reached that IPC losses drop below 5%.[3] Such a design is clearly not appealing from a complexity-effective point of view. Exposing 24 instructions to the select logic, and requiring that all 24 be able to check operand availability each cycle, renders the design not altogether different from a monolithic scheduler.[4] A more compelling approach would be to diminish front-end throughput relative to the number of lanes. For example, a 4-wide front-end dispatching to 8 lanes would reduce the rate at which lanes are consumed by steering relative to the rate at which they are made available by execution. This is precisely what Kim and Smith did in their ILDP work on accumulator-oriented machines [52,53]. My own evaluation of such a configuration yielded a penalty of about 15% — still rather severe, but a substantial improvement over the 30% losses suffered by an 8-wide front-end. I will return to the subject of adding more lanes in Chapter 4, where I explore the factors that determine performance in the more general context of slip-oriented execution.

[3] I should point out that this number is slightly different from the data I will present in Chapter 4 (Figure 4.9 on page 94, specifically). This is because the microarchitecture I use in this chapter differs somewhat from those I use in the more general study presented in Chapter 4.

[4] Each increment in lane count renders the design increasingly similar to a conventional machine, whose 64-entry window can be viewed, abstractly, as a dependence-based one with 64 lanes, each 1-deep.



(a) Static dataflow (b) Monolithic machine (c) Dependence-based

Figure 3.11. Exposing fetch-criticality. The static dataflow on the left shows a hypothetical code fragment comprising two loops. The inner loop has a very low trip count (4 iterations in this example). The program's critical path flows through the outer loop (shaded instructions), bypassing the inner loop entirely. The middle diagram shows dispatch and issue of this code on a 4-wide monolithic machine. Because of its very flexible execution model, the machine is able to fetch and start executing the outer loop's epilogue code (instructions labeled 'E') well before the inner loop completes its execution. On the dependence-based machine on the right, however, the inner loop's dataflow spreads across lanes very rapidly, forcing dispatch to stall several times. Those stalls delay fetch of the critical instructions in the outer loop's epilogue. Note that the inner loop code executes at the same speed on both machines.

Exposing non-critical work as fetch-critical

Implicit in the above discussion was an assumption that the instruction traces under discussion were execute-critical. That is, the stalls imposed by lanes were deemed benign or deleterious depending on their impact on the machine's ability to extract ILP. However, machines do not always operate in an execute-critical regime, and the ILP extracted from a given instruction trace does not necessarily have any effect on runtime. Instruction supply rate is also important and here, too, the impact of lanes can be severe.

Figure 3.11 shows code that is not execute-critical, but whose dispatch stalls impact performance. This example is, in fact, distilled from the `twolf` code sample discussed earlier. It so happens that the small loop traced in Figure 3.8 is nested within another loop that contains critical dataflow on the backward slices of cache misses and branch mispredictions. Figure 3.11(a) depicts this structure. Since the inner loop, though executed

often, has a low and predictable trip count, it constitutes a sequence of instructions that must simply be fetched into the window in order to reach the more important instructions that follow it; the inner loop's execution can safely continue in the shadow of the surrounding, long-latency operations. Figure 3.11(b) shows that a monolithic machine is ideally suited to that requirement, rapidly fetching and buffering the inner loop code and, in the process, reaching the outer loop's epilogue without delay. As a result, the execute-critical dataflow chain in the outer loop is minimally impacted by the presence of the inner loop. By contrast, Figure 3.11(c) shows that diverging dataflow in the inner loop renders it a barrier to forward progress on the dependence-based machine. By stalling on that unimportant code, the machine ultimately delays the dispatch — and hence issue — of the critical dataflow that lies ahead. *It exposes non-critical work as fetch-critical.*

An analogous problem is encountered by `gcc`, which suffers very severe performance losses on the dependence-based machine (recall, again, Figure 3.3). This benchmark spends a significant fraction of its time in a `memcpy` loop, which has been unrolled by the compiler so that each fetch cycle delivers a large amount of ILP into the window. Lanes are therefore consumed very rapidly, and dispatch stalls are commensurately frequent. Because this loop suffers so many cache misses, the throttling of instruction supply in this case translates into a serialization of cache misses and a commensurate reduction in the amount of exposed MLP.

I want to stress that this barrier effect can apply regardless of the IPC achieved within the blocking region. Notice from Figure 3.11 that both the dependence-based and the monolithic machines execute the inner loop code at precisely the same rate — the dispatch stalls are not impacting the machine's ability to extract IPC from the inner loop. But that capability is useless when the code is not itself execute-critical. It is the throttling of instruction supply that matters in this case.

	Original study	My evaluation
Instruction supply	perfect instruction cache. perfect unconditional branch prediction. gshare predictor: 12-bit global history, 4K 2-bit counters.	*same* *same* tournament predictor: 14-bit global history, 16K 2-bit global and choice counters, 2-bit local counters, 12-bit histories.
Front-end	8-wide, unknown depth	*same*, 10 stages to dispatch.
Window	128-entry ROB. 64-entry unified issue queue.	*same* *same*
Execute	8 general-purpose units, 4 memory ports. all instructions unit latency. loads wait for older store addresses.	*same* latencies similar to Alpha 21264. ideal memory disambiguation.
Memory	L1: 32KB, 2-way, 1-cycle hit. L2: perfect, 6-cycle hit.	*same*, 2-cycle hit. *same*, 12-cycle hit.
Back-end	16-wide	8-wide

Table 3.3. Contrasting baseline machine parameters. The second and third columns repeat the information from Tables 3.1 and 3.2, respectively.

3.3 Accounting for experimental discrepancies

Having uncovered the main processes at work in a dependence-based machine, it now becomes possible to explain why my evaluation of the machine differs so much from that conducted by Palacharla *et al.* As I mentioned earlier, in both cases an attempt is made to evaluate the dependence-based scheduler by incorporating it into a machine that stresses the execution core to its fullest. This objective is reflected in each study's choice of baseline machine parameters. Table 3.3 compares those two sets of parameters. It is clear that both simulators model fairly aggressive instruction and data supplies, and both provide ample execute bandwidth. However, a number of differences also stand out.

- *Branch prediction.* Whereas I use an aggressive tournament branch predictor, the original study modeled only a 12-bit gshare predictor.

- *Memory disambiguation.* I modeled ideal memory disambiguation with an unbounded load/store queue, while the original study was more conservative, requiring that loads wait for all prior store addresses to resolve.

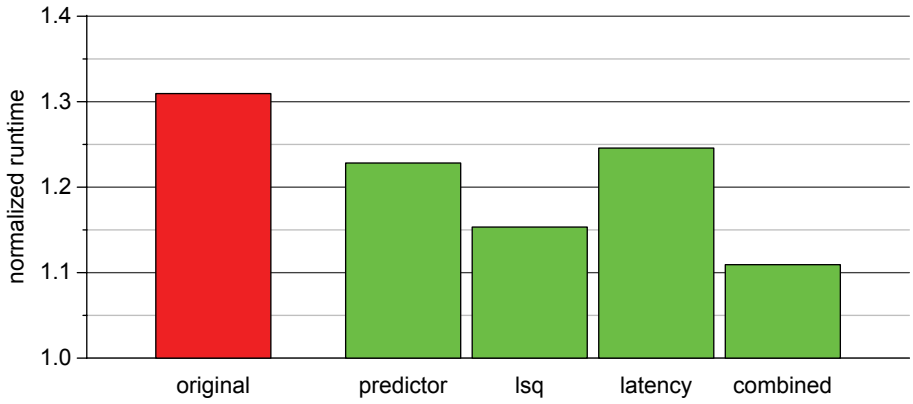


Figure 3.12. The impact of machine parameters. The graph shows how runtime on the dependence-based machine relative to that of the monolithic machine is affected by successive changes to the underlying microarchitecture (of both machines). The bar labeled 'original' represents the configuration I started with in Section 3.1.2 (see Figure 3.3). The 'predictor' bar replaces the tournament branch predictor with the same gshare predictor used in the original study. The 'lsq' bar imposes the original study's memory disambiguation constraint, which prevents loads from issuing until all older store addresses are resolved. The 'latency' bar shows the effect of modeling unit-latency instructions and a smaller L1 miss penalty. The net effect of all these factors is shown by the 'combined' bar. Data is averaged across all the 12 SPEC CINT2000 benchmarks.

- *Latencies.* The original study assumed all instructions execute with unit latency, whereas I used different latencies for different instruction types; L1 cache miss penalties were also different.

I now show that the above differences are, together, sufficient to account for the discrepancy in results between the two studies.

I do so by working backwards from my machine configuration, successively changing the microarchitecture until I reach a configuration similar to the original study.[5] Figure 3.12 shows how relative performance is thereby affected. All microarchitectural changes reduce the penalty incurred by a dependence-based scheduler, so all have a positive effect on the scheduler's apparent efficacy. Among them, the memory disambiguation constraint is clearly the most important, since it alone cuts the relative performance loss in half (from 30% to 15%). Taken together, the various changes reduce a 30% loss to about 11%, bring-

[5] I was unable to discover, both from the literature [75, 76] and from the authors, the size of the load/store queue (LSQ) used to impose the memory disambiguation policy. I chose 32 as a best estimate of the size most likely used in the original study. Palacharla *et al.* used the SimpleScalar Version 1.0 [16] simulation tool in their work, scaling the default RUU size by a factor of 4 to model a 64-entry issue window. Modeling a 32-entry LSQ scales its default size by the same amount.

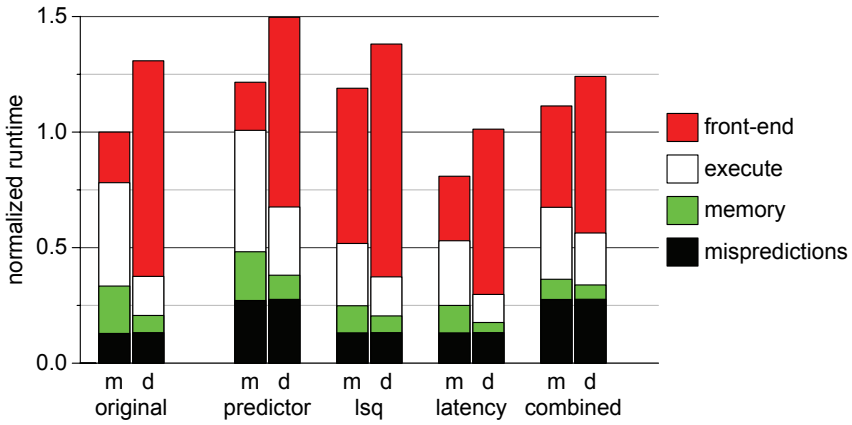


Figure 3.13. The impact of machine parameters on the critical path. Each bar shows how changes in machine parameters (the same as those described in Figure 3.12) affect the critical path breakdown for both the monolithic ('m') and the dependence-based ('d') machines. In this case, all bars are normalized to the leftmost one. Data is averaged across all the 12 SPEC CINT2000 benchmarks.

ing my results close in line with those of the original study.

To understand why relative performance changes in this way, I once again make use of critical path analysis. Figure 3.13 plots the critical path breakdown for the monolithic and dependence-based machines when each of the changes is applied to the underlying machine configuration. Their combined effect (see the rightmost bar in the figure) is to inflict a performance loss on the monolithic machine; IPC drops by about 13%. In contrast, imposing those same changes on the dependence-based machine actually *improves* IPC by about 5%. That the two machines respond so differently is a result of the prevalence of fetch-criticality in one and, equally, its absence in the other. Very broadly, as each change is made to the underlying machine configuration, there is a tendency to increase front-end constraints, thereby shifting the critical path from execute- to fetch-criticality. The dependence-based machine, operating already in a fetch-critical regime, is less affected because those penalties are imposed *in the shadow* of its front-end stalls. Below, I provide a more detailed discussion of how each of the configuration changes impacts the two machines.

Instruction latencies. The primary effect of reduced latencies — particularly of the lower cache miss penalty — is an increase in the rate at which instructions can be executed.

In the monolithic machine, this benefit is dampened by increased pressure on instruction supply, since higher ILP exposes an underlying fetch constraint. That is, there is a *shift* in the critical path from execute- to fetch-criticality: the reduction in time spent on instruction execution and memory operations exceeds the overall reduction in runtime by about 20%. That deficit is accounted for by an increase in time spent waiting for the front-end. The dependence-based machine also benefits from improved execution throughput, but its effect is not to expose an underlying front-end constraint, but rather to alleviate an already dominating one: lanes now clear out quicker, so stalls are resolved sooner.

Branch prediction. Branch mispredictions degrade performance, both through added cycles spent waiting for the pipeline to refill and through the serialization of otherwise parallel work. The first of these two effects is felt equally by the monolithic and dependence-based machines (the 'misprediction' component of the cycle breakdown increases equally for both), but the second effect has very different repercussions. In the monolithic machine, additional execution and memory latency cycles are introduced, and they add directly to runtime — they *lengthen* the critical path. This occurs because mispredictions have a serializing effect, hiding ILP that was previously being exploited. In the dependence-based machine, additional execution and memory cycles are introduced, but they are accompanied by a reduction in fetch-criticality — they *shift* the critical path more than they lengthen it. This occurs because dispatch stalls are now being cleared in the wake of a mispredicted branch, so the additional execution cycles introduced by the misprediction are exposed on the critical path in the shadow of the erstwhile stalls. Thus, while a weaker predictor slows the baseline machine by 22%, the dependence-based machine is slowed by only 14%, reducing the relative performance difference from 31% to about 23%.

Memory disambiguation. The imposition of a conservative disambiguation policy forces loads to wait in the window while older store addresses resolve. This effect manifests itself as an increase in time spent by load instructions in the issue queue and, where those delays

are severe enough to cause the issue queue to fill, as increases in dispatch stalls as well. A more dominant effect, however, arises because of the introduction of a finite-capacity LSQ. It imposes back-pressure in the pipeline when it fills up, thereby throttling instruction supply rate. In the baseline machine, performance drops by almost 20% because of this, and the fetch-critical component of the critical path signature increases dramatically. The dependence-based machine, in contrast, loses only 5% since the LSQ's resource constraint is imposed *in parallel* with extant dispatch stalls. That is, LSQ-induced front-end stalls are *masked* by those already imposed by steering.

3.4 Summary and conclusions

Embedded in the good performance figures originally reported for the dependence-based scheduler is the compelling result that basic properties of dataflow can be exploited to build a very simple scheduler. That is, the original good results would seem to imply that dataflow lends itself very well to a slip-oriented out-of-order execution model. However, my evaluation of the scheduler does not corroborate such a conclusion. I find that the dependence-based machine's steering logic introduces a serious instruction supply problem, thanks primarily to the stalls imposed by the dependence-based steering rules. Those stalls can, in turn, be ascribed to basic properties of dynamic dataflow, which cause instructions to spread "shallow and wide" across the machine's lanes rather than stacking up within them. The net result is that the machine's effective issue queue size is determined by the number of lanes, not by the depth of each one. Indeed, I find that 8 lanes provide the machine with an effective capacity of little more than 16 instructions. Thus is lost one of the most important benefits of a conventional scheduler design: the ability to *buffer* instructions in order to *decouple* dispatch from the order and rate at which instructions are executed. In this respect, a monolithic window is ideal because it facilitates dispatch as long as *any* one of its entries is available; it imposes only its finite capacity as a constraint

on instruction supply. A dependence-based machine, by contrast, renders the window's ability to buffer work subject to the shape of dynamic dataflow.

Diagnosing these problems also points to why the dependence-based scheduler performed so well in its original evaluation. It appears there was a fortuitous interaction between constraints already present in the baseline machine and those introduced by the new scheduler. These caused the scheduler's instruction supply problems to be hidden behind pre-existing bottlenecks. In short, the original study was not exercising the new scheduler, so its performance problems were not exposed.

Given that the original study was conducted about a decade ago, it might be argued that the foregoing merely exposes a problem in mapping the dependence-based design into a more modern context, where simulator configuration parameters are such that the execution core now has a larger impact on overall performance. However, the original study deliberately attempted to model an underlying microarchitecture in which the execution core was stressed to the fullest. The tacit implication, therefore, was that an 8-laned dependence-based machine performs comparably to a 64-entry monolithic one. However, my results show that the real comparison to be made is between an 8-laned dependence-based machine and a monolithic machine with an issue queue of 16 — not 64 — entries. Indeed, using the machine configuration that most closely matches that of the original study (*i.e.* the one used for the 'combined' bar in Figure 3.12), I find that the monolithic machine's performance degrades by less than 8% when its issue queue is reduced from 64 to 16 entries — proof again that the baseline machine used in the original study was over-provisioned in terms of its issue queue buffering capabilities. Viewed in this light, a dependence-based scheduler comprising 8 FIFO buffers (each 8-deep) is not so compelling, from a complexity-effective point of view, given that its performance is no better than a 16-entry monolithic issue queue.

Though the performance problems exposed in this analysis are severe, it cannot be concluded that they are fundamental. After all, it is the steering heuristic — its stalling behavior, in particular — that appears to be at fault; the slip-oriented execution model it-

self cannot be blamed at this point. However, it should be borne in mind that the policy proposed by Palacharla *et al.*, which ensures that only dependent instructions are slotted behind one another in each lane, is important for rendering the in-order issue constraint benign. Any alternative that does not stall will perforce expose instructions to in-order execution. The challenge of improving on the performance of the original dependence-based policy is one to which the next chapter is devoted. There I will show that the dependence-based policy is, in fact, a very good one — a conclusion others have reached following their own failed attempts at finding a better scheme [64].

To conclude, I want to reiterate a few points about empirical technique. My ability to reach all of the foregoing conclusions is underpinned by *critical path analysis*. The critical path dependence graph model I used in this work proved indispensable, both as a tool for characterizing performance (by means of critical path breakdowns) and as a basic conceptual model in which to reason about machine behavior (notions of fetch- and execute-criticality, for example). The first of these has special bearing on the original evaluation of the dependence-based scheduler. An IPC figure, alone, is too coarse a metric for claiming that a particular microarchitectural change is good (or benign) in terms of its performance impact. Though researchers tend to have a feel for what constitutes a reasonable IPC figure, vagaries in experimental setups can cloud this already vague and informal means for reasoning about empirical work. In short, too many aspects of behavior are aggregated into such a number. Sensitivity analysis can help in this regard, but a much more direct and explicit means for analyzing machine behavior is the *critical path breakdown*, which I feel should *complement* an IPC figure. A critical path signature is needed to demonstrate, first, that a given machine does not spend an inordinate amount of time in any one part of the pipeline and, second, that performance does indeed derive from the factors presumed to be responsible. Had a tool for critical path analysis been available in the original study, the dependence-based machine's instruction supply problems would have been immediately exposed.

Chapter 4

Fundamental performance constraints[†]

The previous chapter's results demonstrate that slip-oriented execution, when managed by a dependence-based steering policy, is not effective at extracting ILP from the instruction stream. In this chapter, I explore the potential of slip-oriented execution in a more general context. I will show that it is inherently limited by basic properties of program dataflow. Achieving good performance demands either very sophisticated instruction steering schemes, the complexity of which would probably exceed that of conventional dynamic schedulers; or it demands the use of so many lanes that overheads like global communication would render the design impractical. It appears, therefore, that some form of out-of-order capability is needed at each PE if out-of-order execution is to be effectively synthesized overall.

I start in Section 4.1 by briefly describing the empirical framework I make use of in this chapter. Since I aim to expose severe performance problems even under the best possible circumstances, the simulated machines are devoid of all overheads that arise in the context of a distributed design: the per-lane in-order issue constraint is the only factor constraining instruction execution. In Section 4.2, I then argue that it is basic properties of dynamic dataflow that will pose problems for such machines. That informal analysis is, in fact, a generalization of the previous chapter's exploration of interactions between dataflow and the dependence-based steering policy. Having identified the underlying problems, I then provide the intuition behind two principal avenues for overcoming them.

The first of those, to which Section 4.3 is devoted, is the instruction steering policy. I

[†] The content of this chapter derives from work published at the 14th International Symposium on High-Performance Computer Architecture [86].

show that potential for better steering does indeed exist, but that implementation complexity imposes fundamental obstacles to achieving that potential. In fact, I show that *any* policy aiming to improve performance will *necessarily* be too complex to build. Underpinning that key result is a conceptual framework for reasoning about performance of slip-oriented execution. I use that framework to show that achieving performance within 40% of a comparably-resourced dynamically-scheduled machine requires that each steering decision takes into account its potential to delay execution, both of the instruction being steered and of those yet to be steered. In effect, steering decisions must be governed by exact knowledge of when instructions will eventually execute. It is this requirement that is both necessary for good performance and fundamentally hard to achieve in practice. And it is this requirement that is ultimately imposed by the in-order issue constraint: if steering were instead targeting out-of-order PEs, each decision would have fewer ramifications in terms of when subsequently steered instructions can execute.

The second strategy for improving performance, which I explore in Section 4.4, is adding more lanes to the machine. This is a subject I touched on in the previous chapter (Section 3.2). I will show that increasing the number of lanes relative to instruction fetch width steadily improves the ability of a slip-oriented machine to exploit ILP, even when a simple instruction steering policy is used. However, obtaining acceptable performance demands that lane count exceed fetch width by a factor of 2, a requirement that will exacerbate ILDP-related overheads. Not least of those is the global communication penalty, which I show is liable to outweigh the benefits of having more lanes. I also show, however, that it is principally the addition of extra issue queues, not whole lanes, that facilitates improved performance. This leads me to consider designs with more than one in-order issue queue per lane. Such designs can indeed match the performance of machines with many more lanes, but they rely on a form out-of-order execution at each lane — essentially, a FIFO-based scheduler like that evaluated in the previous chapter — to sustain slip among the local in-order issue queues. As such, these machines approach the capabilities of the

clustered machines, which support conventional dynamic scheduling at each of their PEs.

I want to emphasize that the results described above do not merely show where I was or was not able to extract compelling performance from a slip-oriented machine. Rather, my analysis explores *fundamental* aspects of performance in these machines, independent of artifacts of specific design choices. My principal contribution, therefore, is to delineate the design space, showing where solutions for good performance might be found, and, more importantly, where they simply cannot be found. Though such generality comes at the cost of no specific design proposal, a study of fundamentals is a necessary first step in this area. And though my conclusions are, of course, subject to my assumptions about the underlying execution model, those assumptions are general enough to cover a wide range of designs. Short of fundamental changes to the way a laned machine operates, my principal assertion, which is that some form of out-of-order capability is necessary for synthesizing out-of-order execution overall, holds true.

These results are also important because they have ramifications for CMPs. At the close of this chapter, in Section 4.6, I will show that the generic slip-oriented execution model upon which I base my results is the same one at work in a CMP that can dynamically *fuse* some portion of its constituent in-order cores into a single, large out-of-order processor. Such a machine would be the holy grail substrate for future client system workloads because it would, in principle, cater equally for both TLP- and ILP-rich programs — the latter because coarse-grained parallelism benefits from a large number of simple, low-power in-order cores; the former because ILP-rich codes demand, by contrast, a single, aggressively speculative out-of-order core. However, because a fusing of in-order cores constitutes an implementation of slip-oriented execution, my results paint a rather negative picture for the prospects of such machines. Sustaining good single-thread performance in a CMP context appears, therefore, to demand out-of-order capabilities in at least some of the CMP's constituent cores.

4.1 Background

In this short section, I describe the empirical framework I use in this chapter. As I noted earlier, my focus is on basic principles, not on specific designs, so I idealize the simulated machines to the point that only the underlying slip-oriented execution model differentiates the laned machine from a conventional out-of-order superscalar. I also present initial performance figures in Section 4.1.2 to confirm that slip-oriented execution is indeed prone to severe performance problems. Since this involves the use of the dependence-based steering policy, the results of that analysis are very similar to those presented for the dependence-based scheduler in Chapter 3. I repeat them here because the simulated machines in this section are slightly different from the dependence-based scheduler: the whole execution core is now partitioned, not just the issue queue. That is, adding lanes to the machines simulated in this chapter now has the effect of increasing net issue bandwidth, not just issue queue buffering capacity.

4.1.1 Empirical framework

Throughout this chapter, I will evaluate two types of laned machine, one with a 4-wide front-end and one with an 8-wide front-end. For each of those, I explore configurations in which the in-order lanes support 1- and 2-wide issue. For convenience, I refer to the resulting configurations using the nomenclature Fw–($L\times I$w), where F denotes the machine's effective front-end width, L the number of lanes it has, and I the issue width of each of those lanes. I use the shorter $L\times I$w notation when front-end width is clear from context. I do not, in general, require that $F = L \cdot I$. More specifically, I permit configurations in which the number of lanes exceeds the front-end width. For example, the 4w–(8×1w) machine fetches 4 instructions per cycle, but steers them among 8 1-wide lanes.

In all cases, the laned machines I examine are constrained only by their slip-oriented execution model; all other overheads arising in the context of a distributed execution model

are ignored. I therefore make the following optimistic assumptions about laned machine implementation.

1. *Front-end.* The laned machines have the same front-end pipeline as an equal-width monolithic machine (so as to model the effects of register rename). Instruction steering, no matter how complex a policy being modeled, introduces no extra latency.

2. *Execution core.* Global communication is free: transferring values among lanes occurs with zero latency and infinite bandwidth. Memory disambiguation occurs in a centralized manner, and introduces no overheads no matter how many lanes are present. Each lane's issue queue has unbounded capacity, so only the ROB imposes a limit on the number of in-flight instructions.

3. *Memory.* I assume a single data cache for all lanes, and that the cache has enough bandwidth to support any number of lanes. This is particularly optimistic since I will be exploring machine configurations in which the number of lanes is 8 or more, requiring that the memory system be able to support 8 or more simultaneous accesses.

I will, in some cases, model a global communication penalty, but only so that I can quantify the extent to which the idealized assumptions are benefiting performance. In general, these optimistic assumptions are important given the rather negative conclusions I draw from this study: that performance results are poor in spite of them adds weight to my claim that the performance potential of laned machines is inherently poor. Furthermore, modeling a very generic laned machine facilitates generalizing the results of my analysis. In particular, I show in Section 4.6 that the performance limits have ramifications for fusion of in-order cores in a CMP.

To gauge the efficacy of the laned machines, I compare their IPC to that of 2-, 4- and 8-wide out-of-order superscalar designs, aiming thereby to understand where in the spectrum of out-of-order performance capabilities the laned machines reside. As usual, I will refer to

	2-wide	4-wide	8-wide
Instruction supply	perfect instruction cache; perfect uncond. branch pred.; tournament cond. branch pred.: 14-bit global history, 2-bit local counters, 12-bit histories.	same.	same.
Front-end	2-wide, 5 stages to dispatch	4-wide, 8 stages to dispatch	8-wide, 12 stages to dispatch
Window	64-entry ROB; 32-entry issue queue.	128-entry ROB; 32-entry issue queue.	256-entry ROB; 64-entry issue queue.
Execute	2 int., 2 floating pt; 1 mem. latencies similar to Alpha 21264; ideal memory disambiguation.	4i / 4f / 2m. same.	8i / 8f / 4m. same.
Memory	L1: 64KB, 4-way, 2-cycles; L2: 8MB, 8-way, 12-cycles; DRAM: 300 cycles.	same.	same.
Back-end	2-wide.	4-wide.	8-wide.

Table 4.1. Monolithic baseline machine configurations.

those baselines as the *monolithic* machines. Table 4.1 enumerates their main architectural parameters.

4.1.2 Performance evaluation

To confirm my assertions that laned machines are prone to poor performance, I now compare the performance of a few basic configurations against that of the monolithic machines. To manage instruction steering in the laned machines, I once again make use of the *dependence-based steering policy*, which I explored at length in the previous chapter. Though I have already exposed inherent problems with this scheme, it turns out to be a remarkably good policy for laned machines. Indeed, one of my principal objectives in this chapter is to demonstrate that outperforming dependence-based steering is fundamentally hard. I should nevertheless point out that there are a number of other steering policies presented in the literature — and later in this dissertation — that have been shown capable of delivering very good performance. But they are not suitable for laned machines: they are targeted specifically at machines implementing the distributed dataflow-oriented execution

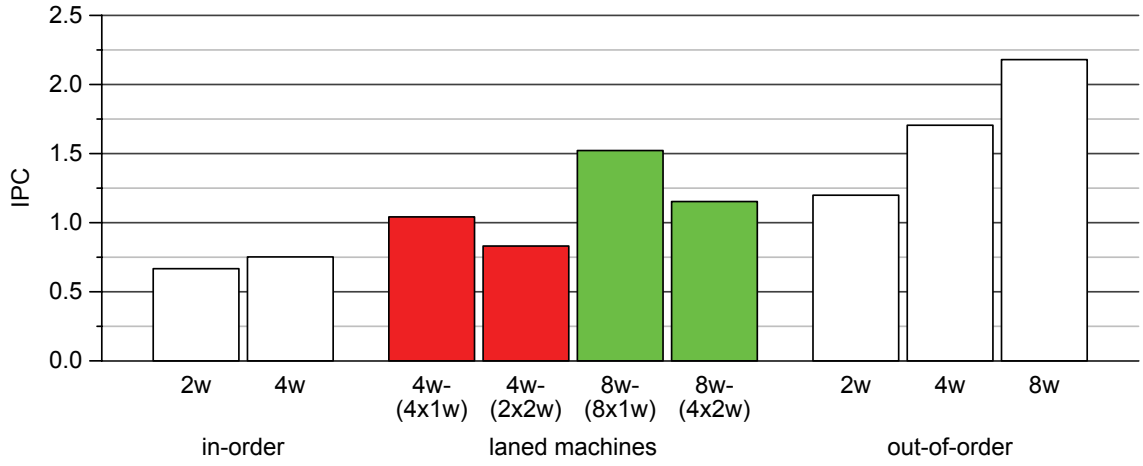


Figure 4.1. Laned machine performance. Each bar shows harmonic mean IPC across the SPEC CINT2000 benchmarks for a variety of machine configurations: 2 in-order superscalars, 4 laned machines, and 3 out-of-order superscalars. Table 4.1 enumerates the configuration parameters for the out-of-order machines. The in-order machines are configured similarly in terms of their memory hierarchy and issue capabilities, but their front-end pipelines are shorter.

model, where each PE supports out-of-order execution. The best among them exploit instruction criticality to guide their choices [35, 84]. They can do so because out-of-order execution affords them a great deal of flexibility in terms of *focusing* resources on critical instructions. When PEs impose an in-order issue constraint, however, there is little room for movement in this regard. In fact, I have implemented in-order variants of the (very effective) policies to be presented in the second half of this dissertation, and find that they perform no better than the basic dependence-based policy. In short, the dependence-based policy remains the best published for distributing instructions among in-order execution units, and serves therefore as a good starting point for this chapter's analysis.

I implemented the dependence-based steering policy for four basic laned machine configurations: the 4w–(4×1w) and the 8w–(8×1w) machines, which comprise 1-wide in-order PEs; and the 4w–(2×2w) and the 8w–(4×2w) configurations, which have 2-wide in-order PEs. Figure 4.1 compares their performance to that of various monolithic designs. The laned machines improve on the performance of superscalar in-order designs, but they fail to match the out-of-order machines. Both the 4-wide and 8-wide laned machines fall

well short of an equal-width monolithic out-of-order design, being about 1.6 and 1.4 times slower, respectively. In fact, the laned machines perform more like a much smaller out-of-order machine. Both of the 4-wide laned machine configurations fail to match even the 2-wide out-of-order machine; only one of the 8-wide laned machines manages to do so, but both fall short of the 4-wide out-of-order machine.

4.2 Understanding the challenges

Since I do not model any other distributed execution model overheads, it is specifically the per-lane in-order issue constraint that is to blame for the above performance results. In this section, I informally describe the key problems this constraint introduces and, in so doing, provide the intuition behind the two avenues for its mitigation that I will be exploring in the remainder of the chapter.

4.2.1 Dataflow chains

A per-lane in-order issue constraint interacts poorly with very basic properties of dynamic dataflow. In general, dataflow is not uniform in shape: not all instructions execute with unit latency, and ILP is distributed unevenly in the instruction stream. As a result, sustaining an IPC of N generally requires simultaneously managing more than N chains of dataflow in the window. Figure 4.2 shows this effect at work in a fragment of the `gap` benchmark. Immediately after cycle 1, there are 10 *live dataflow chains* — 10 instructions which have no dataflow predecessors in the window. Not all of those are ready to issue at this time, so the achievable IPC is well below 10; but they will soon become ready. In a dataflow-oriented execution model, each live instruction will be able to issue as soon as it becomes ready, since it is only dataflow that constrains its execution. The same effect can be achieved in a slip-oriented model only if each live instruction reaches the head of an issue queue by the time it is data ready. That, in turn, demands either having enough lanes to buffer all

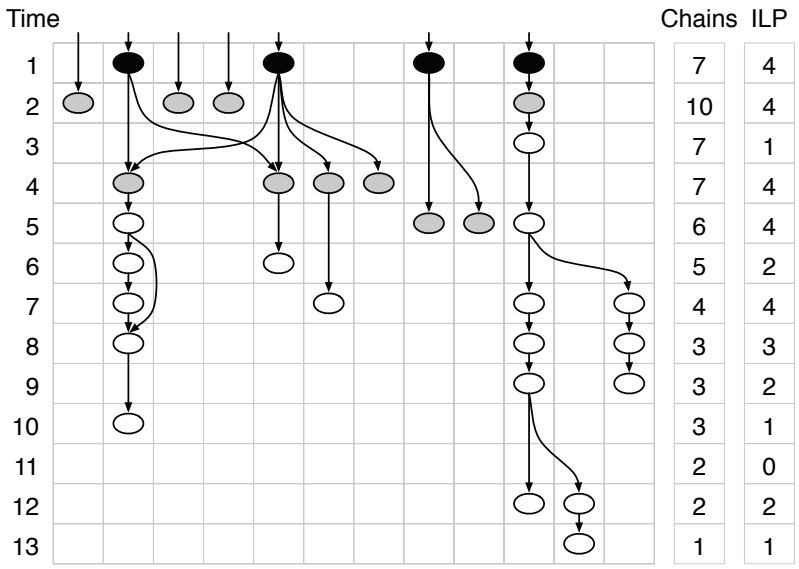


Figure 4.2. Dataflow comprises many short chains. The diagram shows the schedule induced by the 4-wide monolithic machine on a fragment of the `gap` benchmark. All instructions were resident in the machine's issue queue at the start of the timing, and left it at the indicated issue times. The lightly shaded instructions constitute the set of *live chains* active in the window after the issue, in cycle 1, of the dark instructions.

the live chains; or it demands carefully interleaving the live chains among the available lanes. The former option is an expensive prospect given the data in Figure 4.3. The graph shows the cumulative distribution of cycle-by-cycle counts of live chains active in the issue queues of the 4- and 8-wide monolithic machines. If chains are to reside at distinct lanes, the number of lanes required will not be small. The 4-wide machine, for example, would need 14 or more lanes if it is to have sufficient buffering more than 90% of the time. The second option — interleaving dataflow chains among the available lanes — is a patently hard problem: the right interleaving must be effected at instruction dispatch time, well before each instruction's execution time is known. In the next subsection, I make these ideas more concrete by way of an illustrative example.

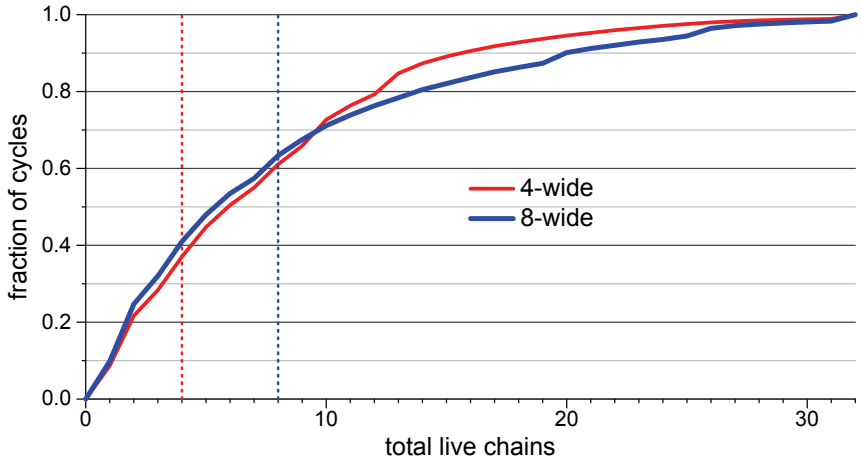


Figure 4.3. Cumulative distribution of live chain counts The graph plots the cumulative distribution of cycle-by-cycle counts of the live-chains active in the issue queues of the 4- and 8-wide monolithic baselines. The vertical lines highlight the fraction of the distribution that would be covered by 4 and 8 lanes. Data is averaged across all 12 SPEC CINT2000 benchmarks.

4.2.2 An illustrative example

Figure 4.4 uses a hypothetical code example to distill the key factors that underlie the performance problems encountered by laned machines on real code. The diagram shows the execution of a small loop on machines with 2-wide fetch and execute bandwidth. Assuming that no branch mispredictions occur, the peak performance achievable on that loop is 1.6 instructions per cycle. The timing diagram in Figure 4.4(b) shows that a conventional out-of-order machine can sustain that execution rate. The flexibility afforded by its dataflow execution model is key to its ability to do so: instructions are able to execute as soon as their operands are ready; the order in which they enter the window, and their location in the issue queue, has no bearing in this regard. By contrast, the laned machine shown in Figure 4.4(c) performs much worse because its in-order issue constraint serializes ILP in the window. Specifically, instruction `D` (and hence `F` and `G`) in lane `L1` is prevented from executing when its operand is ready because it is blocked while instruction `C` waits for its operand from `A`. The net effect is a complete serialization of successive loop iterations, reducing sustained ILP to just above 1.

A more judicious allocation of instructions to lanes can overcome these problems. It

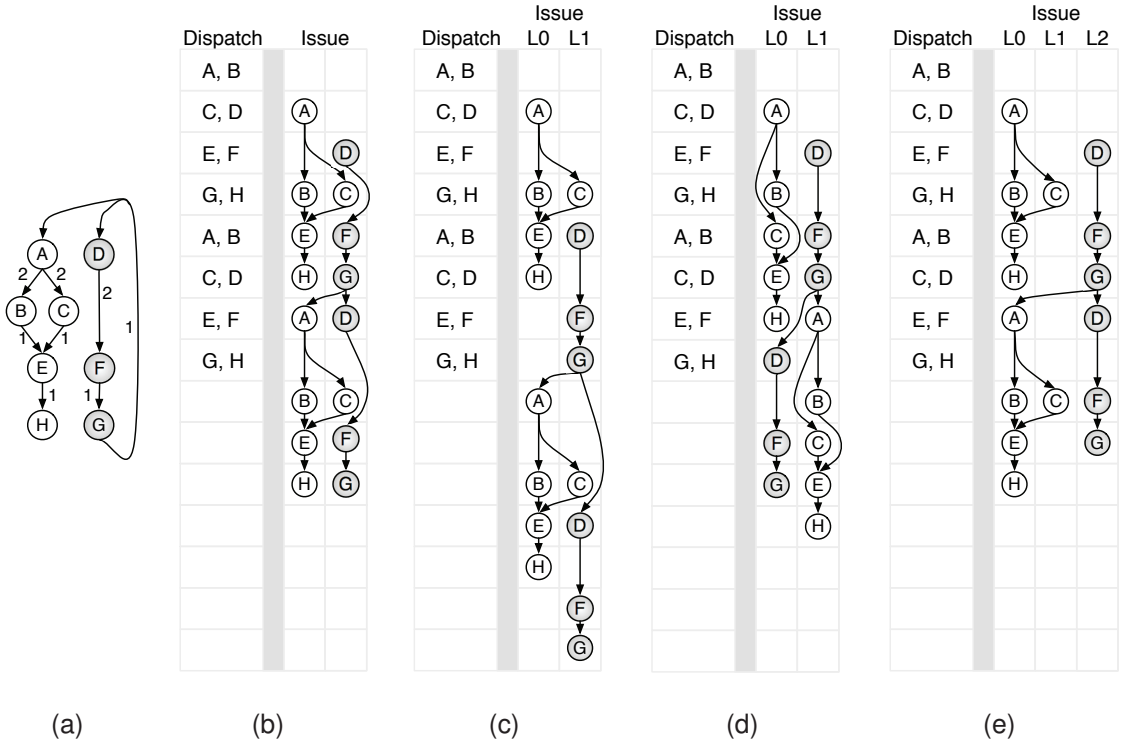


Figure 4.4. The impact of lanes. The static dataflow diagram on the left shows the body of a small, hypothetical loop. Dataflow edges are labeled with the latency of the producing instruction. The timing diagrams — (b) through (e) — show dispatch and issue of two successive iterations of the loop on various machines, all of which have a 2-wide front-end and 2-wide issue bandwidth. In (b), the schedule obtained by a conventional out-of-order machine is shown. Diagram (c) shows the operation of a 2w–(2×1w) laned-machine when instructions are steered using a dependence-based policy similar to that described in Chapter 3 (Section 3.1). Diagram (d) shows the operation of a more sophisticated steering policy that interleaves live dataflow chains. Diagram (e) shows the operation of the same dependence-based policy used in diagram (c), this time with an additional lane made available to it.

is the allocation of instruction `C` to lane `L1` in Figure 4.4(c) that is responsible for the performance losses, since it is this instruction that consistently blocks the `D`–`G` chain. The schedule shown in Figure 4.4(d) shows how steering instruction `C` to `L0` exposes the ILP available via the `D`–`G` chain. Although the `A`–`H` chain now takes slightly longer than necessary to execute, the resulting IPC matches that of the out-of-order machine. This example makes it clear that steering can have a profound impact on performance, but also that good steering decisions are not necessarily obvious. Indeed, the decision made in Figure 4.4(d) required knowing, in advance, that instruction `C` should be deliberately delayed in order to make room for instructions `D` through `G`. Even harder, it required knowing that the delay

imposed on C in sending it to lane L0 would be small enough not to outweigh the benefits of doing so. I will show in Section 4.3 that any steering policy that can effectively interleave live chains in this manner will be too complex to build.

An alternative means for improving performance is shown in Figure 4.4(e). In this case, the addition of a third lane to the machine permits even a simple steering policy to match the IPC of the monolithic machine. This is possible because the D–G dataflow chain can be steered in such a way that it is no longer exposed to the in-order issue constraint. It is not the added issue bandwidth that helps in this case, however, but rather the improved ability to expose the live chains to the issue logic: adding lanes increases the chances that an instruction will be at the head of a lane when it becomes data ready. Of course, adding more lanes to a machine is not a very cheap or efficient means for improving its ability to exploit ILP, especially when all the overheads inherent to a partitioned design are taken into account. I explore the relationship between lane count and performance in Section 4.4.

4.3 Instruction steering

In this section, I focus on machines with a modest number of lanes. My objective is to determine if it is possible to develop a steering scheme capable of outperforming the dependence-based policy evaluated in Section 4.1.2. I tackle this problem with the benefit of hindsight, having already explored a large number of alternative heuristics, all of which failed to do any better. In the process, I very soon reached the conclusion — as did others [64] — that this is not a problem for which a simple heuristic will suffice. In this section, I put that informal claim on a rigorous footing. In Section 4.3.1, I introduce an abstract model for reasoning about fundamental requirements for making good steering decisions. That model, which is agnostic to microarchitectural details and to steering policy specifics, permits me in Section 4.3.2 to distill the key features required of *any* policy that aims to effectively distribute instructions among in-order issue queues. I then show in

Section 4.3.3 that those features render such policies too complex to build. In fact, their operation would amount to a form of out-of-order scheduling, but of a strictly harder variety than is implemented in out-of-order processors.

4.3.1 Reasoning about steering

As I noted above, my focus in this section is on machines with a modest number of lanes. By this I mean machines whose lane count matches the front-end width, and hence whose aggregate issue width is close to that of a comparable monolithic machine. Recalling the data in Figure 4.3, this means I will be exploring machines whose lane count is lower than the number of live chains typically active in the window. For the sake of expediency, I will restrict the discussion to the 4w–(4×1w) and the 8w–(8×1w) machines; the results that follow can easily be extended to machines with 2-wide lanes.

My framework for reasoning abstractly about steering is founded on a simple observation: any steering policy induces an *instruction schedule* when it distributes instructions among the lanes, since sending an instruction to a lane implicitly determines the time at which it will execute. The matrix in Figure 4.5 depicts this idea. It shows the state of a steering-induced schedule immediately before instruction i is about to be steered. The progress of time is captured by the rows of the matrix (I adopt the convention that time flows downward), and the machine's lanes are modeled by matrix columns. Each cell in the matrix therefore represents a single issue slot (assuming each lane is a 1-wide execution unit). Since lanes impose in-order issue, each column is constrained to receive instructions in sequential order, but there is no constraint on the order of execution across the columns.

The line $h(i)$, which I call the (issue) *horizon* for instruction i, represents the earliest possible time at which i will be able to issue. This is a function both of the time at which i's operands will be ready (denoted $data(i)$) and the time at which i is dispatched into the window (denoted $disp(i)$). Specifically, $h(i) = max\{disp(i), data(i)\}$. In addition to its horizon, execution of i is constrained by the current state of the matrix: the most recently

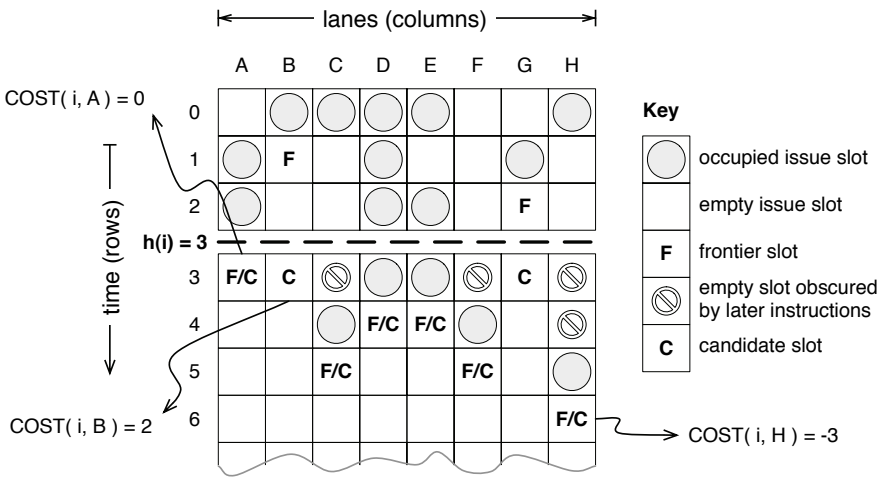


Figure 4.5. A scheduling matrix. The matrix captures the state of a laned machine at each cycle, in this case immediately before instruction i is about to be steered. Rows of the matrix capture the progress of time (time flows downward), and columns represent the machine's lanes. Assuming each lane is a 1-wide machine, each cell in the matrix constitutes a single issue slot.

occupied issue slot in each column imposes a lower bound on when instruction i will be able to execute at that column. This is of course an artifact of the in-order issue constraint at each lane. I call the first available issue slot at column c the *frontier* of that column, and write $f(c)$ to denote that slot. Each column's frontier is marked with an 'F' in Figure 4.5.

There are only 8 possible slots into which instruction i can be placed, one in each column. I call these the *candidate slots* for i; each is marked with a 'C' in Figure 4.5. All of these occur no earlier than both the horizon for i and the frontier of the corresponding column. Thus, if $cs(i, c)$ denotes i's candidate slot in column c, then $cs(i, c) = max\{h(i), f(c)\}$.

Different steering choices will have very different repercussions for instruction i and for the ensuing state of the matrix, and hence also for subsequently steered instructions. These effects can be captured by means of a *steering cost* metric:

$$COST(i, c) = h(i) - f(c)$$

When the column to which i is steered is clear from context (or not important), I will simply write $COST(i)$.

The cost metric will be negative when an instruction becomes ready *before* a column's frontier, in which case $COST(i)$ represents the number of cycles beyond its ready time that instruction i is delayed. The candidate slot in column `H` in Figure 4.5 is a case in point. I say a steering decision incurs *internal cost* in such cases, thus reflecting the notion that it is the instruction itself that pays the penalty. By contrast, decisions for which the cost metric is positive will be said to incur *external cost*, since these are liable to penalize subsequently steered instructions. A positive cost implies an instruction becomes ready only *after* the column's frontier, so sending it there effectively *hides* a number of issue slots from other instructions. Sending instruction i to column `B`, for example, would incur external cost. The candidate slot in column `A` is an example of a zero-cost steering decision: instruction i is not delayed beyond its horizon, nor is any earlier issue slot wasted by sending it there.

Note that external cost is entirely an artifact of the per-column in-order issue constraint. Steering an instruction to an out-of-order PE cannot incur external cost because issue slots are not thereby obscured from subsequently steered instructions. And it is precisely the potential for external cost that renders a good policy for laned machines fundamentally hard to obtain.

4.3.2 Requirements for good steering

I now show that an effective steering policy must pick columns by taking $COST(i)$ into account and, more importantly, that it must do so from *both* an internal (negative) and an external (positive) cost point of view; it is *not sufficient* to take just one of them into account.

Internal cost

I take it as self-evident that paying heed to internal cost is crucial to ensuring good performance.[1] If there is *no* constraint on how often steering costs are internal, then there will be no constraint on the extent to which instructions are delayed beyond their issue horizons. This is an easy claim to verify. A policy which *randomly* allocates instructions to lanes, for example, places no bound on the potential for negative-cost steering decisions. My implementation of such a scheme incurs average slowdowns in excess of 70% relative to the monolithic baselines.

The dependence-based steering policy is, in fact, an example of a scheme that takes internal cost into account. To understand why, it is useful to reconsider the details of how it operates. Recall from Section 3.1 that the policy uses register dependences among instructions to try slot consumers immediately behind the producers of their operands. Specifically, it sends an instruction to a non-empty lane if (one of) its producer(s) immediately precedes it there; if no such lane exists, it sends the instruction to an empty lane, perhaps first stalling until one becomes available. In collocating an instruction directly behind its producer, the policy guarantees that the instruction's issue will be delayed only by that of its producer — a constraint already imposed by dataflow. And in sending instructions to an empty lane, it likewise guarantees that only operand availability will constrain instruction issue. These rules establish a simple invariant: instructions wait in the issue queue only for their operands; the in-order issue constraint imposed by each lane is never exposed, being always subsumed by dataflow constraints. More precisely, if instruction i is steered to lane ℓ, then $COST(i, \ell) \geq 0$. I call this the *internal cost invariant*.

The dependence-based policy's stalling behavior is instrumental to maintaining this invariant. It is through stalling that the policy ensures that instructions are never steered behind independent instructions, and hence that they wait only for their operands once

[1] More accurately, internal cost ought to be *minimized* for critical path instructions and *bounded* for non-critical instructions. In both cases, however, internal cost must be taken into account.

they are dispatched. It is easy to confirm the importance of stalling — and thereby further confirm that internal cost is important — by evaluating schemes in which instructions that would otherwise cause the dependence-based policy to stall are instead steered to a lane picked by some heuristic. This amounts to occasionally slotting an instruction behind independent work. One of the best-performing heuristics I found picks the least-full lane, an indicator that it might drain soon, and therefore that internal cost might be low. It performs substantially worse than the basic dependence-based policy, increasing performance losses by a further 12% on the 4-wide configurations and by more than 35% on the 8-wide machines.

Of course, stalling is not itself benign, as I showed in detail in the previous chapter. However, a simple experiment suffices to show that stalls are, in fact, just a manifestation of a more fundamental problem. I implemented an oracular version of the dependence-based policy, one that never stalls, but which instead sends an instruction to the same lane that the original policy would otherwise have stalled and waited for. That is, the oracular policy knows ahead of time which lane will drain first. Nevertheless, it performs *no better* than the basic dependence-based policy, which means stalls are hiding a more basic problem with dependence-based steering. That problem is external cost.

External cost

The dependence-based policy guarantees that its steering costs are zero or positive, never negative. In a sense, it bounds the cost of its decisions *from below*. But it cannot impose any bounds from the other direction: the extent to which steering decisions become largely positive — that is, external — is not controlled. In fact, Figure 4.6 shows that it is liable to frequently pick the slot that maximizes external cost. When it cannot collocate a consumer with a producer, it picks an empty lane — precisely the one whose frontier is likely to be the furthest away from the consumer's issue horizon (instructions are not always data-ready when they enter the window). Thus, while choosing an empty lane is key to avoiding nega-

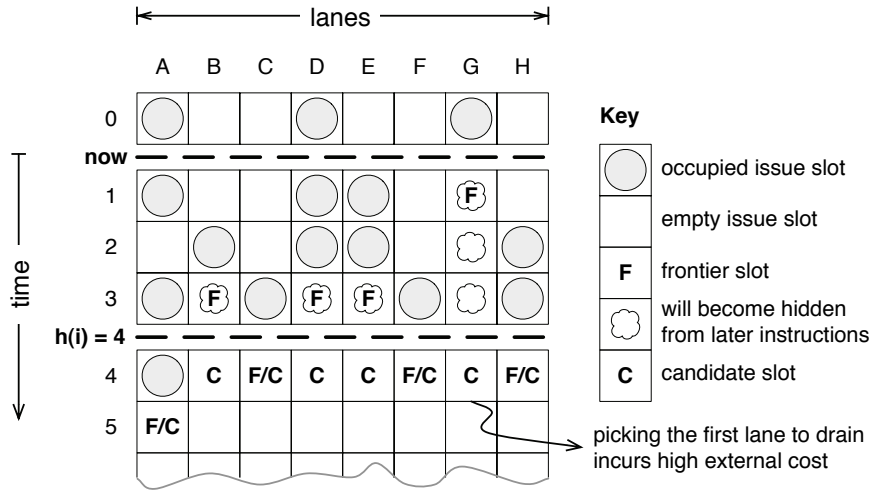


Figure 4.6. The problem with dependence-based steering. At cycle 1, steering logic is about to dispatch instruction i. If i does not depend on a value live at any frontier, it will be sent to column G, which is the one that has just drained. This is precisely the choice with maximum external cost.

tive cost decisions, doing so tends also to push cost into the positive dimension. Enforcing the internal cost invariant amplifies external cost.

At the root of this problem is the fact that the dependence-based scheme has no means by which to take external cost into account. Dataflow dependences, alone, are not sufficient because bounding the cost of steering decisions *from above* necessarily requires knowing the frontier of each column. And that demands knowing when the last instruction in each lane will issue.

Optimizing both internal and external cost

The previous two sections argued that ignoring either the internal or the external dimension of steering costs is deleterious to performance. That is, taking both dimensions into account is *necessary* for good performance. I now show that doing so is also *sufficient*. I do so by examining an idealized steering policy in which I equip the steering logic with prescient knowledge of the state of execution at each lane, including exact knowledge of which loads will hit in the cache. In short, I give the steering logic a precise view of the matrix

its decisions are inducing. I defer for the moment a discussion of the implementation challenges such a policy would face in practice.

Knowing the state of the matrix permits the idealized policy to directly compute and optimize the cost metric for each of its decisions. For each lane ℓ, it computes $COST(i, \ell)$, then it picks, from among those that have a positive cost (*i.e.* no internal cost), the one whose value lies closest to 0; if no such lane exists, it picks the one whose negative value is closest to 0. Such a policy will tend to optimize $|COST(i)|$ since it tends to avoid slots with extreme internal or external cost. But it gives higher precedence to small positive values than it does to small negative values, meaning it will always prefer penalizing other instructions over penalizing the current one. It is greedy in this respect.

Figure 4.7 (bars labeled 'ideal') shows this cost-aware policy delivers much better performance than the dependence-based one (bars labeled 'dep-based'), reducing an initial 60% slowdown in the 4-wide configuration to about 15%, and an initial 30% slowdown in the 8-wide machine to about 6%. The 8-wide machine benefits the most because its higher lane count affords the policy more flexibility in optimizing the cost metric. The 4-wide machine offers little opportunity in this regard, especially, for example, when the forward slice of a cache-missing load blocks a lane, thereby consuming 25% of the machine's issue resources.

These performance figures are not the best that an idealized policy can do. Indeed, I have implemented more sophisticated (less greedy) policies in which internal and external cost considerations are weighted differently based on the criticality of each instruction. Although these policies perform even better than the one I present here, I do not show data for them because doing so would add no weight to my main claim, which is that taking both dimensions of the cost metric into account is sufficient for substantially improving performance. Moreover, a more sophisticated heuristic will be even more complex to implement; and, as I now show, the greedy policy is already too complex.

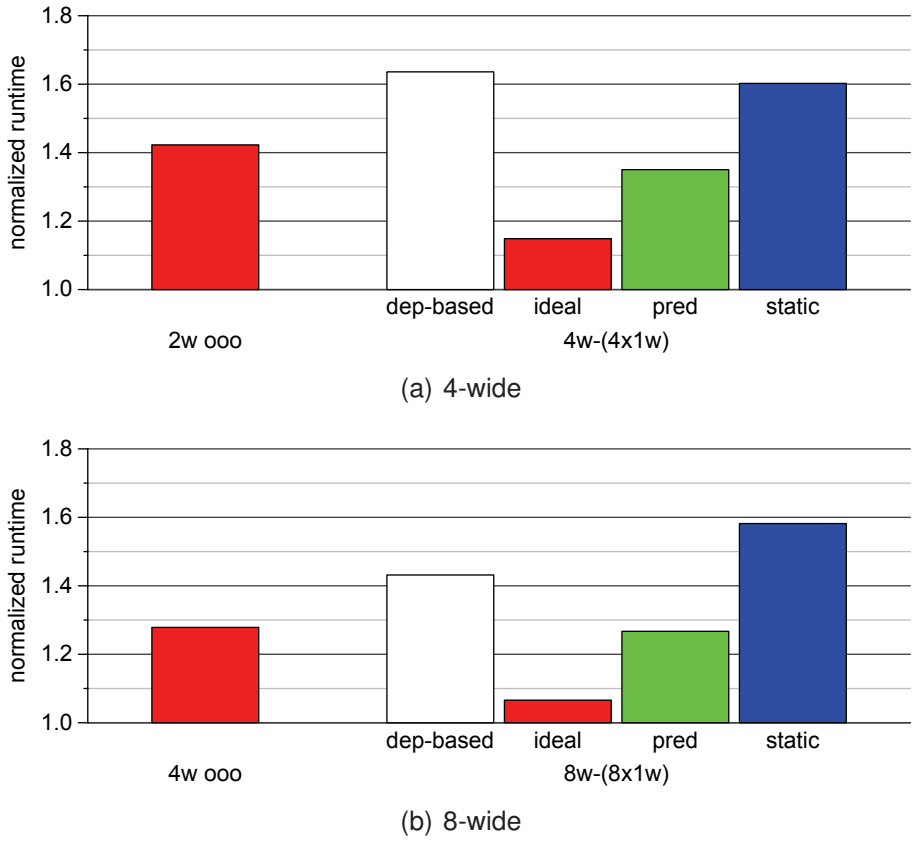


Figure 4.7. Optimizing both internal and external cost. The top graph plots runtime of various machine configurations relative to that of the 4-wide monolithic baseline (lower is better); the bottom graph shows performance relative to the 8-wide monolithic baseline. The '2w ooo' and '4w ooo' bars in each graph show, for reference, the performance of a monolithic machine half the size of the baseline. The remaining bars show performance of different steering policies for the laned machine: the idealized cost-optimizing policy ('ideal'); that same policy with oracular knowledge of load latencies replaced with predictions from a load hit/miss predictor ('pred'); and then finally with a static latency prediction in which all loads are assumed to hit in the cache.

4.3.3 Implementing cost-aware steering

Implementing the idealized policy — or, indeed, any policy that takes both the internal and external dimensions of cost into account — quickly runs into a number of problems. The culprit, of course, is external cost. As I noted earlier, internal cost can be tackled simply by taking dataflow dependences into account, but targeting external cost, by definition, involves measuring the impact a steering decision will (or might) have on instructions that are not necessarily data dependent on it. Specifically, it demands having some idea of where (in time) each lane's frontier resides, as well as where an instruction's issue horizon lies

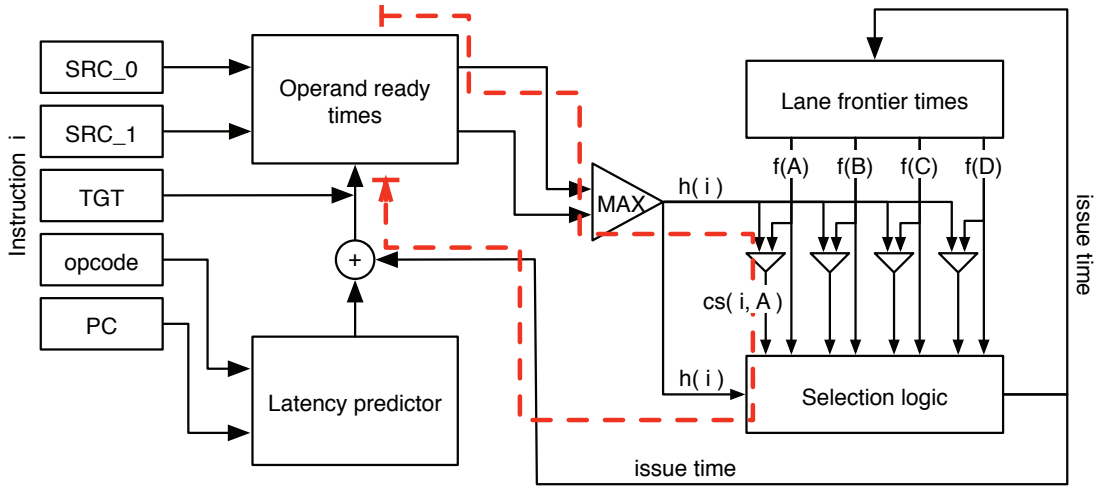


Figure 4.8. Cost-aware steering. The diagram shows the main components involved in making a cost-aware steering decision for instruction i on a laned machine comprising four lanes (named A through D). The selection logic computes $COST(i,\ell) = h(i) - f(\ell)$ for all lanes ℓ, then picks the lane that yields cost closest to 0. The (expected) issue time for i is given by $cs(i,\ell)$, where ℓ is the lane selected for i. This value is used to update the lane frontier times and, together with predicted execution latency, the operand availability time for i's target register. The dashed line running through the diagram identifies the critical loop induced by this feedback.

relative to that. Both of these are inextricably linked to how dataflow in the lanes will be executed. The challenge is knowing and using that information *before* execution actually occurs — when steering decisions are made. This points to the need for an approach similar to the *dataflow prescheduling* techniques previously proposed for wakeup-free schedulers [28,64]. These maintain information on when each in-flight instruction is expected to execute, and hence when its results will be available.

In terms of the idealized policy, a prescheduling approach would have to operate as follows when steering instruction i; Figure 4.8 shows the steps graphically.

1. *The issue horizon.* Interrogate in-flight operand state to determine the time at which each of i's source operands will be ready. The horizon, $h(i)$, is the maximum of the current time ($disp(i)$) and the time at which operands will be ready ($data(i)$).

2. *Candidate slots.* For each lane ℓ, use $h(i)$ and $f(\ell)$ to compute $cs(i,\ell)$, the earliest possible time at which i will be able to issue at ℓ.

3. *Pick a candidate slot.* From among the lanes ℓ for which $COST(i, \ell) \geq 0$, pick the one that minimizes external cost. If no such lanes exist, pick the one that minimizes internal cost.

4. *Update state.* If ℓ was picked at the previous step, update $f(\ell)$ to $cs(i, \ell) + 1$. Also update operand state to record the fact that the output register of i will be available at time $f(\ell) + latency(i)$, where the latter term denotes the expected execution latency of instruction i.

Thus expressed, two problems immediately come to the fore. The first pertains to the computation of $latency(i)$ for variable-latency instructions like loads. Returning to Figure 4.7, the 'pred' and 'static' bars show that performance rapidly degrades when one compromises a little bit on the accuracy of load latency information, and hence on frontier information. The 'pred' bars show the impact of replacing oracular knowledge of load latency with a load hit/miss prediction [101]; those labeled 'static' show the effects of assuming all loads hit in the cache. In the former case, both the 4- and 8-wide machines lose most of the performance won by the idealized information; both are now performing at close to the level of the smaller monolithic machine. When static instruction latencies are assumed, performance drops back to the levels of dependence-based steering and, in the case of the 8-wide machine, to even worse levels.

A more fundamental problem with a prescheduling approach is exposed when one considers its superscalar implementation. The problem is that the steering decision for one instruction might depend on the outcome of its predecessors in the same fetch packet. This is analogous to the problems faced by register rename, and indeed by dependence-based scheduling, but now the problem is more acute. Simply put, prescheduling of multiple instructions cannot be performed in parallel, nor pipelined across cycles, because the state that seeds each decision (operand readiness and frontier information) is not available until *after* each steering decision is completed. This is in contrast to register rename logic, which permits back-to-back fetch packets to be renamed over successive cycles because the results

of one packet (the new destination tags) are available after a single cycle. The same results in prescheduling (the destination columns) are not available until a scheduling decision for each instruction has been made. The dashed line in Figure 4.8 delineates this critical loop. In this respect, the prescheduling logic is more complicated than the wakeup-select logic used in out-of-order machines, since the latter does not have to schedule dependent instructions in the same cycle.

4.4 Adding more lanes

I mentioned in Section 4.2.1 that sustaining an IPC of N generally demands managing more than N live dataflow chains at the same time. Since only the heads of each issue queue are visible to the issue logic, effective management of those chains in a laned machine equates to ensuring that each reaches the head of an issue queue by the time it is ready to issue. The previous section showed that doing so is not practicable when there are fewer lanes than there are typically live chains. The only remaining strategy, then, is to forego smart steering and to equip the machine with enough lanes — specifically, enough lane heads — to buffer all the live chains. I show in Section 4.4.1 that the dependence-based steering policy, which naturally ensures that live chains end up at distinct lanes, benefits as expected from the addition of lanes. But it manages to match monolithic performance only when lane count is high. Moreover, this applies only under my idealized assumptions about ILDP-related overheads. If I take into account global communication, for example, which is inevitably exacerbated by the addition of more lanes, those performance gains are quickly overwhelmed. The picture is not entirely negative, however. I show in Section 4.4.2 that supporting multiple in-order issue queues per lane, together with some modifications to the issue logic, yields almost the same performance as many lanes.

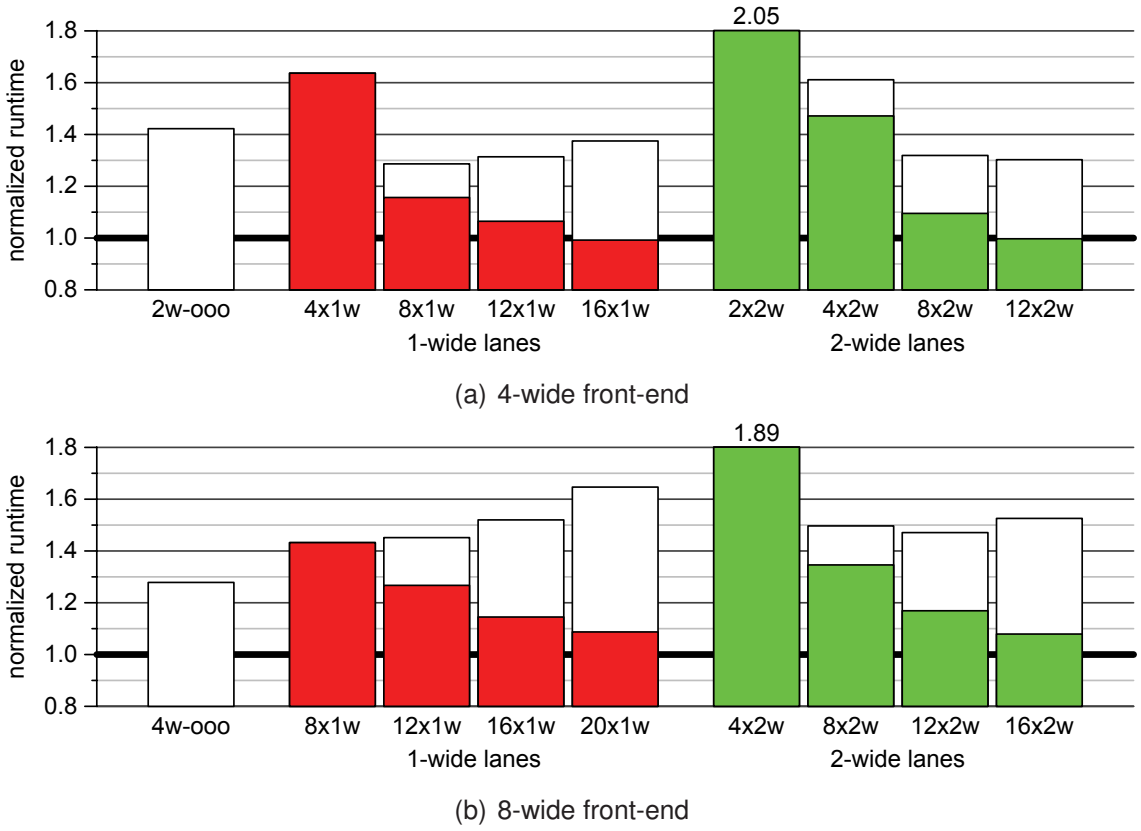


Figure 4.9. More lanes help performance. Each graph shows harmonic mean runtime (lower is better) of various laned machines, first with a 4-wide front-end (top) and then with an 8-wide front-end (bottom). Values in the top graph are normalized to the runtime of the 4-wide monolithic machine; those in the bottom, to the 8-wide monolithic machine. For reference, each graph shows the runtime of a smaller monolithic machine (the '2w-ooo' and '4w-ooo' bars). The shaded portion of each bar shows performance in the absence of ILDP-related penalties. The unshaded portion shows additional slowdowns that would be incurred if a global communication penalty is imposed: 2 cycles are added for every 4 lanes added to the narrowest (leftmost) configuration.

4.4.1 Performance from more lanes

The dependence-based steering policy slots instructions directly behind a producer, or to an empty lane if that is not possible. Live chains will therefore always end up at distinct lanes, a property which renders dependence-based steering a natural candidate for exploring the potential of a machine with many lanes. Figure 4.9 shows its performance as a function of increasing lane counts. Average performance losses relative to the 4-wide monolithic machine drop below 10% only at the 12×1w and the 8×2w configurations. In other words, we would need to have more than twice as many lanes as front-end width before perfor-

mance of a 4-wide monolithic machine is matched. Note that these results are consistent with the data in Figure 4.3, which showed that covering more than 90% of the live chain distribution would demand 14 or more lanes.

I should point out that the benchmark `mcf` is an outlier for machines with a 4-wide front-end. For most benchmarks, the laned machines manage to match the 4-wide monolithic baseline by the time they have 8 lanes; `mcf` lags monolithic performance no matter how many lanes the machine has. While the average results look a lot better if `mcf` is ignored, this anomaly highlights an important problem encountered by laned machines, one I mentioned in the previous chapter: the laned machines are ill-suited to exploiting memory-level parallelism. When a cache miss occurs, any lane holding consumers of that miss will become blocked until the data returns from memory. The `mcf` benchmark suffers from high cache miss rates, so it tends to consume lanes rapidly, and to block them for a long time. A monolithic machine is ideally suited to dealing with these situations, as the comparatively good performance on `mcf` attests. That the other benchmarks perform well on the laned machines by the time they have 8 lanes can be attributed, therefore, to their very good cache miss behavior. It should also be borne in mind that the 8-lane machines have an aggregate memory bandwidth of 8 memory operations per cycle, whereas the 4-wide monolithic machine can issue only 2 memory operations per cycle. This artificially boosts the apparent efficacy of the laned machines. A more realistic implementation, which would take into account the overheads incurred by supporting such a high memory bandwidth, would substantially reduce the apparent efficacy of an 8-laned machine.

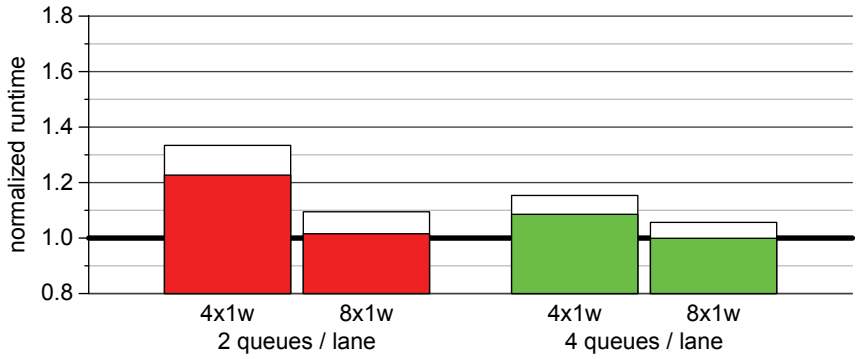
The above results are optimistic, of course. At a minimum, they assume that inter-lane communication is free. One could perhaps argue that four 1-wide lanes could be floor-planned to achieve inter-operation forwarding as efficiently as a 4-wide monolithic machine, but it is hard to believe the same holds for eight or more lanes. The unshaded portions of the bars in Figure 4.9 show the relative execution time when global communication penalties are introduced, 2 cycles for every 4 additional lanes beyond the fetch width of the

machine. With a two-cycle forwarding latency, performance of the 4w–(8×1w) machine now lags almost 30% behind that of a 4-wide monolithic processor; and modest increases in that latency very quickly counter the benefits of adding more lanes. Because a larger number of lanes exacerbates other complexities of building a laned machine — among them, the crossbar for steering fetched instructions to lanes, bandwidth to the memory system, and static and dynamic power consumption — it is difficult to build a compelling case for such machines.

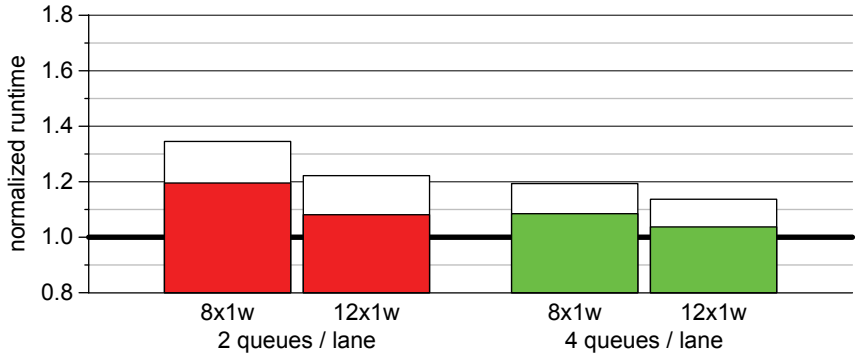
4.4.2 Multiple issue queues per lane

The benefit of many lanes derives ultimately from the availability of issue queue slots for live chains, not from the extra issue bandwidth. One potentially appealing design point, therefore, is to share issue resources among 2 or more issue queues. This would provide steering logic with a sufficient number of target issue queues and, at the same time, would mitigate the problems of a design that is too highly partitioned. But issue logic would have to evolve from a simple in-order scheme to one capable of selecting ready instructions from more than one issue queue. A natural candidate for such a scheme is the dependence-based scheduler I examined in Chapter 3. That is, the laned machine would be modified so that each PE uses a dependence-based scheduler, rather than in-order issue logic, to pick ready instructions from among the heads of two or more issue queues. Each PE would therefore implement a modest form of dynamic scheduling, doing so in order to achieve the requisite degree of slip among its local issue queues — slip that previously arose naturally through the decoupled nature of execution at each lane.

Figure 4.10 shows the effects of adding a dependence-based scheduler to 1-wide in-order PEs, each now equipped with more than one issue queue. Performance remains very close to that of buffer-equivalent laned machines. The 4×1w machine, for example, when equipped with 2 FIFO buffers per lane, almost matches the basic 8×1w machine; and 4 FIFO buffers brings it close to the original 16×1w machine's performance. These



(a) 4-wide front-end



(b) 8-wide front-end

Figure 4.10. Multiple issue queues per lane. Each graph shows the effect of having more than one in-order issue queue at each lane. In the top graph, each pair of bars shows the results for laned machines with 4 and 8 1-wide lanes, first with 2 issue queues per lane, then with 4 per lane. The bottom graph shows the analogous data for machines with an 8-wide front-end, this time with 8 and 12 lanes, each with 2 and 4 issue queues per lane. The unshaded portion of the bars shows the effect of a 2-cycle global communication penalty. All bars show harmonic mean runtime (lower is better) relative to a 4-wide (top) and 8-wide (bottom) monolithic machine.

results do not conclusively prove that a slip-oriented execution model is a viable alternative to conventional dynamic scheduling, but they do show that a modest number of lanes, together with reasonably modest enhancements to the issue logic, can mimic the benefits of many lanes. That said, it is not clear that a dependence-based approach at each PE is really a complexity-effective alternative to full out-of-order capabilities, especially if those capabilities involve just a small issue queue and narrow issue width. Moreover, from a performance potential point of view, out-of-order PEs are indeed very compelling, as I will show in Part II.

4.5 Generalizing the results

In this section, I show that the foregoing analysis, being built upon a very general and an idealized view of laned machines, will have ramifications for a certain class of CMPs. In Section 4.5.1, I briefly provide some background context to establish a link between laned machines — as I have been describing them so far — and the notion of *horizontal fusion* of in-order cores on a many-core substrate. I then show in Section 4.5.2 that these fused designs, when stripped of all fusion-related overheads, look no different than the idealized laned machine I have been examining to this point. The previous observations about fundamental performance limits therefore apply equally to this idealized view of fused-core designs.

4.5.1 Background

Trends in client system workloads point to the need to support two classes of program in the future. The first comprises massively-parallel, compute-intensive programs such as graphics, physics, signal processing, and recognition, mining and synthesis (RMS). The second constitutes the large body of remaining applications that resist parallelization, be it via manual or automatic techniques. Programs in the first class benefit from a wealth of narrow, perhaps multithreaded, cores of modest clock speed, wide SIMD functional units and a high-bandwidth memory system; those in the second need a single, high-frequency, wide-issue, aggressively-speculative, out-of-order processor. A key challenge facing architects today, therefore, is designing a single chip that supports both classes of program equally well.

One proposed solution is the heterogeneous CMP [55,56]. This comprises a small number of aggressive out-of-order superscalar cores, plus a sea of small, efficient, throughput-oriented cores. Each type of core is designed specifically for the class of workload it is intended to execute, but this approach has two drawbacks. First, a chip vendor must now

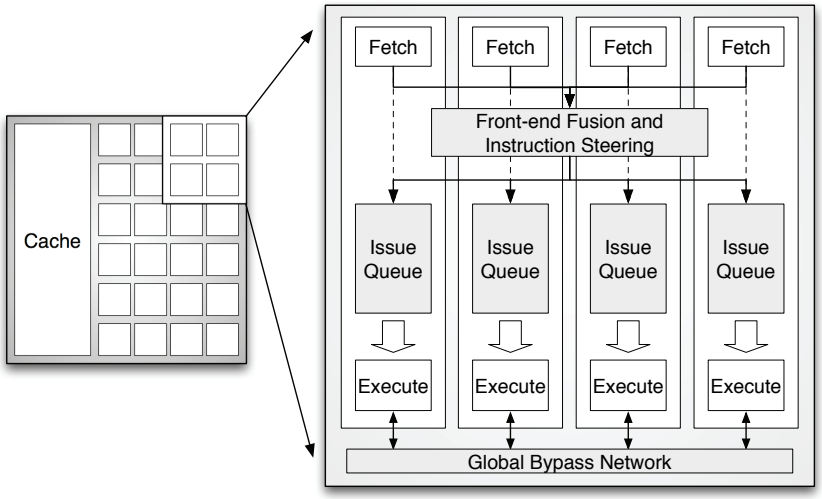
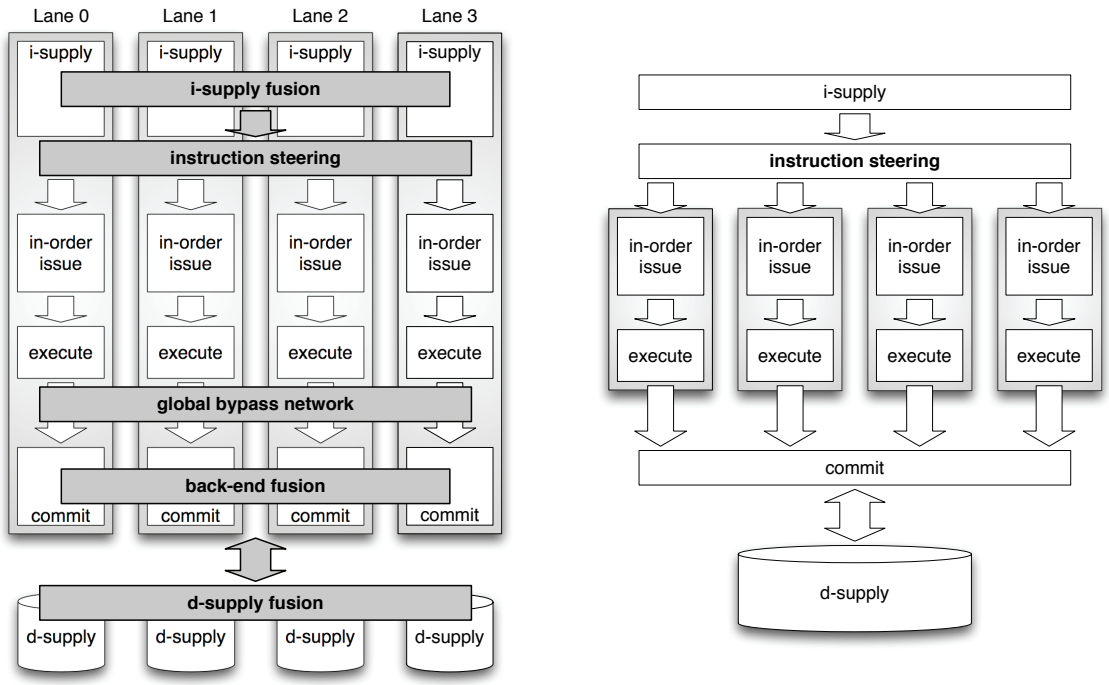


Figure 4.11. Fusion of simple in-order cores. The homogeneous CMP on the left comprises many simple cores, subsets of which (4 in this case) can be fused into a distributed, wide-issue processor to support sequential workloads.

design two (or more) processors for a single product release. Second, any applications that fall between the two extremes on the workload spectrum may be poorly served.

An alternative solution is depicted in Figure 4.11. This uses a homogeneous many-core CMP, which is extended to support on-demand *aggregation*, or *fusion*, of small processor cores into a large out-of-order uniprocessor [44, 51]. Its principal benefit is that it requires the design of just one simple core, yet is flexible enough to cater to a wide range of TLP and ILP in the workload. Though recent research into such designs has focused on fusion of small dynamically-scheduled processors, all of the many-core designs being developed or announced by industry, such as Sun's Niagara [96], Intel's recently announced Larrabee, ClearSpeed's CSX600 [22], the Cell's SPEs [46], Ageia's PhysX [4] and GPUs, are in-order substrates. This is a reflection of the target domain for those machines, where parallel workloads are the norm. Out-of-order execution is not required for those programs, neither for scheduling (since narrow machines are used), nor for latency tolerance (since multithreading can be used instead). In fact, supporting out-of-order execution would only incur area and power overheads, reducing the chip's peak throughput potential.

The ideal design, therefore, would be a many-core substrate comprising just in-order



(a) Fusing in-order cores (b) Idealized machine

Figure 4.12. Horizontal fusion of in-order cores. The diagram on the left shows a high-level view of four in-order cores (lanes) being fused into a larger, 4-wide machine. New hardware required to coordinate the fused cores is shaded. The diagram on the right shows an idealized view for these machines, one in which all overheads introduced by the fusion-specific hardware are ignored.

cores, but which supports on-demand *fusion* of those cores to synthesize out-of-order execution capabilities when they are needed. In such a framework, each core in the fused machine would execute instructions in-order, but execution across cores would be decoupled. Clearly, this is a form of slip-oriented execution.

4.5.2 Horizontal fusion in-order cores

Figure 4.12(a) provides a high-level view of a machine that horizontally fuses four small in-order cores. When fused, the individual cores operate together as a single processor. In the front-end, each core contributes to the fused machine's fetch bandwidth by supplying a portion of the fetch packet each cycle. Coordinating these otherwise disjoint instruction supply units requires the addition of a centralized structure, shown in the diagram as the

front-end fusion logic. This potentially complex piece of machinery controls instruction fetch from distinct instruction caches, manages control flow prediction in some way, and orchestrates fetch redirection when branches are predicted (detected) to be taken (mispredicted) at one of the cores. Once each fetch packet has been assembled, the front-end logic renames instructions and then steers them to an issue queue at one of the cores, where they will eventually execute.

Because cores are in-order, an instruction becomes eligible for issue when it reaches the head of its issue queue and when all its operands are available. Some of those operands will perhaps have been computed remotely at another core, so some form of inter-core communication mechanism must be provided to distribute data among the cores. Figure 4.12(a) shows a global bypass network for this purpose. In this respect, the fused machine is clearly no different from ILDP designs.

Instructions that have completed execution have their results committed, in program order, by centralized back-end fusion logic. This is needed because, like laned machines, execution across the cores does not occur in lockstep, so instructions can execute out-of-order. The set of instructions in-flight between dispatch and retirement therefore constitute an instruction window, no different, in principle, from that maintained by any out-of-order machine. And the execution model supported within that window, being locally in-order and globally out-of-order, is clearly slip-oriented.

The additional hardware required to coordinate otherwise independent cores inevitably introduces a number of overheads. For example, fusion logic necessarily lengthens the front-end pipeline, thereby increasing the branch misprediction penalty, and inter-core communication is exactly analogous to global communication in ILDP machines. However, if those overheads are ignored, the machine starts to look very much like the high-level laned machine depicted in Figure 2.1 (page 41). Specifically, the following assumptions can be made (*cf.* assumptions about laned machine overheads described in Section 4.1.1).

1. *Front-end.* Capturing just the effects of register rename, the fused machine's front-

end can be idealized to operate no differently from an equal-width monolithic machine. Thus, coordination of control flow across lanes and instruction steering logic is assumed to introduce no additional overheads.

2. *Execution core.* Inter-core communication costs can be ignored. Likewise, memory disambiguation can be assumed to occur in a centralized manner and without any additional delays. Issue queues can be assumed unbounded.

3. *Memory hierarchy.* Each core has access to the data cache of any other core without additional latency, and that all caches can support whatever bandwidth is required by the fused design.

Figure 4.12(b), depicts the idealized machine that arises as a result of these very optimistic assumptions.[2] Clearly, this is no different from the laned machines I have been modeling throughout this chapter. Given that those machines are inherently limited in terms of their ability to exploit ILP, and given that this idealized view of a fused-core machine embeds even more optimistic assumptions about overheads than does the model I have been using up to this point, it is clear that a fusion of in-order cores is no more likely to deliver good performance than is a laned machine.

4.6 Summary and conclusions

The idea that out-of-order superscalar performance might be synthesized from in-order scalar parts is an extremely compelling one. It would offer an opportunity to design simple, power-efficient components, yet sustain the aggressive ILP processing capabilities currently provided by dynamically-scheduled machines.

In this chapter, I explored the prospects of such designs by evaluating the performance potential of *slip-oriented out-of-order execution*, the underlying execution model that char-

[2] I should point out that, general though this view is, it does not cover a fused design like Voltron [102], which relies fundamentally on the compiler to orchestrate when and how fusion occurs, as well as to statically map instructions to lanes. I confine my discussion here to schemes that are invisible to the compiler.

acterizes these machines. Despite its conceptual appeal, I find that basic properties of dynamic dataflow fundamentally constrain that model. Specifically, matching the performance of monolithic machines requires the ability to monitor, and ensure prompt execution for, many simultaneously-active chains of dataflow, many more than the average ILP extracted from the program. As a result, modest designs, which use a reasonable number of PEs, and which employ implementable instruction steering schemes, exhibit best-case IPC performance that is well below that of smaller out-of-order machines — designs we are able to build, at modest complexity, already. It is, in principle, possible to do better with more sophisticated steering policies, but the complexity thereby introduced turns out to be no better than conventional out-of-order issue logic; indeed, it appears to be worse.

Configurations with a large number of in-order PEs have good performance potential, however. Though the overheads introduced by having to manage many PEs would render such designs impractical, it is the ability to buffer instructions at in-order issue queues, not the ability to sustain very wide instruction issue, that is the principal benefit of adding PEs. As a result, supporting multiple issue queues per PE is an appealing alternative design. But this would require some modifications to the issue logic at each PE to permit each to sustain the requisite degree of slip among the now local queues. Dependence-based scheduling is the natural candidate for this purpose.

By making some very optimistic assumptions about overheads incurred in fusing in-order cores, all of the above results map naturally into a CMP context. This is important since the ability to horizontally fuse in-order cores, and thereby effectively exploit ILP in single-thread programs, would render a CMP comprising many, simple in-order cores an ideal candidate for supporting a diverse range of workloads in client systems. That laned machines are not effective at exploiting ILP therefore does not bode well for the prospects of such designs. That said, the fact that multiple issue queues per PE work well in a laned machine might hold promise in a CMP context. This is because many-core substrates (*e.g.* Sun's Niagara) typically support *multithreading* at each in-order core, so there is al-

ready a requirement for multiple instruction buffers at each core. If the introduction of a scheduler like the dependence-based one does not constitute too complex an addition to these in-order cores, there is potential for good performance through fusion of a modest number of cores. Further analysis of the various trade-offs is needed, for which purpose a more specific design than the general model I evaluated here would be needed.

Part II

The Clustered Machines

Chapter 5

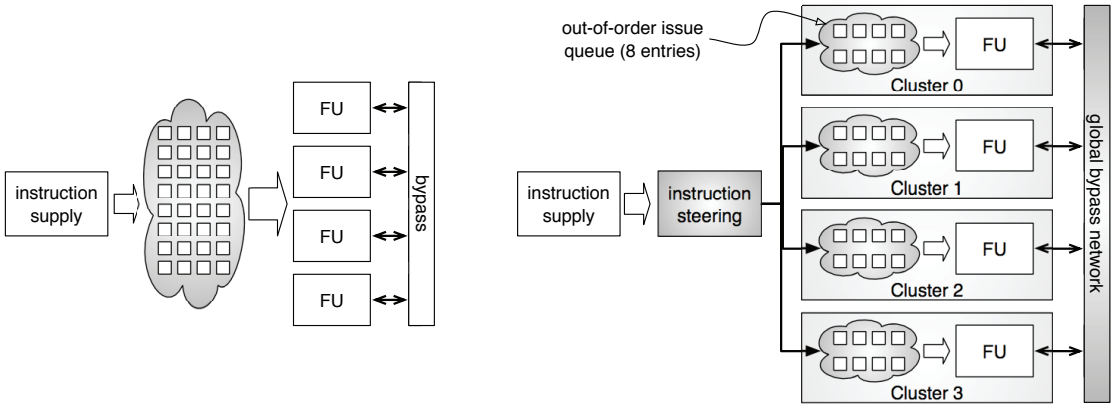
Introduction

Figure 5.1 shows how a 4-wide monolithic machine with a 32-entry issue queue can be partitioned into 4 1-wide PEs. Associated with the 8-entry issue queue at each of those PEs is a dynamic scheduler, no different from that used by the monolithic machine, only now smaller. These machines therefore implement what I call a *distributed dataflow-oriented execution model*. In line with previous work in this area, I call these designs the *clustered machines*, and refer to their out-of-order PEs as *clusters*.

I will show in this part of the dissertation that supporting out-of-order execution locally renders the clustered machines inherently superior to their laned counterparts. In short, the reason is that each steering decision no longer constitutes a scheduling decision as well, since sending an instruction to a cluster no longer constrains it to execute after all instructions already there. The degree of complexity required of steering logic is commensurately less. This is not to say instruction steering in clustered machines is a trivial problem. Indeed, prior work in this field would suggest that clustered machines are inherently ineffective at exploiting ILP, being necessarily constrained by global communication penalties and resource contention stalls. The work I present in this part of the dissertation will counter that view.

5.1 Outline

The first important result I will present is that the distributed and monolithic dataflow-oriented execution models are, in principle, equally capable in terms of finding and ex-



(a) 4-wide monolithic, 32-entry IQ

(b) 4-wide clustered machine, 4x8-entry IQs

Figure 5.1. A clustered machine. (*cf.* Figure 2.1, page 41.) A 4-wide monolithic machine (left) housing a 32-entry out-of-order issue queue is partitioned into a 4-PE clustered machine (right). Each PE is itself a 1-wide out-of-order execution unit with an 8-entry issue queue.

ploiting ILP in the instruction stream. Put another way, the communication and contention constraints that are introduced by the distributed model do not constitute a fundamental barrier to matching the performance of the monolithic one. I present this result in Chapter 6. I show, by means of an idealized scheduling infrastructure, that it is possible to distribute a program's dataflow among out-of-order PEs without, in the process, extending overall execution time. The idealized scheduler exploits oracular knowledge of the *critical path* through a given code region to distribute instructions in such a way that the most performance-critical are the least likely to be exposed to execution constraints; only the least critical — those with the most *slack* — incur penalties. Ultimately, this means critical path dataflow tends to be narrow enough to execute without delay at even the narrowest clusters. Equally, it means non-critical dataflow is able to shoulder execution model constraints, if those constraints are judiciously imposed. It also implies that the IPC penalties reported in previous clustered machine studies are artifacts of specific instruction steering and scheduling policies, not of limits inherent to the distributed execution model.

Having demonstrated that performance potential is good, I then show in Chapter 7 how to go about exploiting that potential in practice. I start by using *critical path analysis* to un-

cover the main causes for performance losses observed by a state-of-the-art framework for managing instruction steering and scheduling in clustered machines. This leads to the development of three new policies for mitigating performance losses. All of them are based on a new criticality metric, *Likelihood of Criticality (LoC)*, which explicitly records the frequency at which a given static instruction appears on the critical path. This is a more fine-grained notion of criticality than conventional binary (critical/not critical) measures, permitting distinction to be made between instructions that are frequently critical, but not always at the same rate as one another. I show how this new metric can be used as the basis for novel criticality-aware steering and scheduling policies that, together, yield performance on 2-, 4- and 8-cluster machines that is within 2%, 4% and 6%, respectively, of an equally resourced monolithic machine.

Chapter 8 zooms in on the new LoC metric. I show that it is generally a stable property of a program, in the sense that a given static instruction's LoC value tends not to vary within and across program runs. As a result, LoC lends itself to offline analysis. To that end, I develop a novel scheme for computing LoC using *profile-guided random instruction trace generation*. This relies on hardware profiling support already available in modern microprocessors to collect profile information for individual instructions. Using the aggregated data from such profiles in conjunction with a program binary, short (1,000 instruction) program traces are fabricated by tracing through the code, randomly evaluating branches in proportion with the collected profile. I use those randomly-generated traces to perform offline critical path analysis, the results of which I show are just as accurate as those produced by an online hardware critical path predictor. A key to achieving this accuracy is correctly identifying memory dependences in the fabricated trace, for which purpose I demonstrate a new approach that uses abstract interpretation to identify memory dependences without explicitly profiling them.

This offline scheme for computing LoC is a first step toward a more complete offline infrastructure for managing clustered machine steering and scheduling policies. In Chap-

ter 9, I present an initial exploration of a *mostly-static dynamic machine*, a co-designed framework in which the bulk of policy logic is incorporated into an offline software component, but one in which hardware retains enough dynamic decision making capabilities to respond to runtime events that cannot be anticipated statically. Since my work in this area is exploratory, I do not present a complete solution for this framework. My principal aim, rather, is to discover the extent to which the right steering and scheduling decisions are both statically discernible and dynamically stable. In this respect, the results I obtain are encouraging. I show that a comparatively simple *offline instruction aggregation* scheme, which statically groups instructions into larger entities so that hardware will steer all of them to the same cluster, is able to match the performance of a fully-dynamic scheme. At the same time, having moved a large fraction of the policy logic out of hardware, the dynamic component of the system is vastly simplified. Overall, this initial study shows that a software-assisted clustered machine is a promising avenue for further research. I conclude Chapter 9 with a discussion of some of the more important questions still to be tackled in this area.

Before starting with the idealized study of inherent performance potential, I briefly describe now the empirical framework that I use in this part of the dissertation.

5.2 Empirical framework

I will consistently measure the performance of various clustered machine configurations relative to an 8-wide monolithic baseline. I choose such a large (*i.e.* wide) baseline because 4-wide designs with very fast clocks and modest power consumption are already being built by chip vendors. An 8-wide machine, by contrast, is considered too large for a monolithic design, rendering it an ideal baseline for evaluating the potential of a clustered microarchitecture. The clustered machines I evaluate are configured so as to apportion the monolithic machine's execution resources equally among the clusters. I therefore examine

Instruction supply	perfect instruction cache; perfect unconditional branch predictor; tournament conditional branch predictor: 14-bit global history, 2-bit local counters and 12-bit histories.
Front-end	8-wide, 12 stages to dispatch.
Execute	256-entry ROB, 128-entry unified issue queue; 8 integer, 8 floating point; 4 memory ports; latencies similar to Alpha 21264; perfect memory disambiguation.
Memory	L1: 64KB, 4-way, 2-cycles; L2: 8MB, 8-way, 12-cycles; DRAM: 300 cycles.

Table 5.1. Baseline (monolithic) machine parameters.

three cluster configurations: two 4-wide clusters, four 2-wide clusters and eight 1-wide clusters, henceforth referred to as the 2x4w, 4x2w and 8x1w configurations, respectively. I will often refer to the monolithic baseline as the 1x8w machine. Table 5.1 summarizes its main architectural parameters. The clustered configurations are identical in all respects, except that, as noted, the execution resources are divided equally among the clusters. For example, the 4x2w configuration partitions the monolithic machine's 128-entry issue queue into four 32-entry queues at each cluster. Likewise, the 8-wide execution bandwidth is divided equally among the four clusters, so that each can issue 2 instructions per clock (no more than 2 integer operations, 1 floating point operation and 1 memory operation per cycle).[1] All clusters load from/store to a shared L1 data cache.

In any clustered architecture, global communication latency and bandwidth are important design parameters. In the experiments I present here, my focus lies on machines with a 2-cycle global communication latency, a reasonable figure for machines with between 2 and 8 clusters. I will also present data for a 4-cycle penalty to show how various schemes are liable to scale with more pessimistic assumptions about inter-cluster latency. In terms of bandwidth, I assume the global bypass network has enough capacity to support peak execution rates: I do not model contention for communication slots. Nonetheless, I do monitor

[1] I round up partial resources, so each cluster in the 8x1w machine has one memory port.

inter-cluster communication and find that, on average, the various schemes I develop incur between 0.1 and 0.25 global values per instruction for the 3 clustered machine configurations I study. These values are no larger than those exhibited by previously published schemes.

Chapter 6

Idealized potential

Research into clustered designs has tended to focus on implementation mechanisms and comparative studies of various steering policies [6, 8, 12, 19, 32, 35, 49, 76, 77, 98]. While these are of course important issues, focusing exclusively on them leaves open the fundamental question of how well those mechanisms and policies exploit the underlying hardware to its fullest. This is an important subject to address, for two reasons. First, it underpins the tenability of clustered microarchitectures as a potential solution to the implementation problems suffered by monolithic designs. If, like their laned counterparts, the clustered machines necessarily suffer performance penalties, and those penalties are as large as they presently appear to be, the whole design space does not look very appealing. Second, even if certain penalties are inherent, it is important to know what they are in order to gauge the success of different proposals, and hence to justify effort being directed at finding better solutions.

This chapter explores the inherent performance potential of clustered machines. In essence, this is really a question about program dataflow and, more specifically, about how dataflow interacts with the constraints imposed by the distributed dataflow-oriented execution model. As I pointed out early on in this dissertation (in Chapter 1), the critical path has a central role to play in this question, for the simple reason that a distributed model's constraints will impact performance only to the extent that they are imposed on the critical path. More to the point, they will impose no overall performance loss if they can be successfully shifted from the critical path onto just the non-critical instructions. For this to be possible, dataflow must be of a shape that lends itself to optimizations that give

preferential treatment to critical code. Specifically, the following must hold.

1. *Critical dataflow must be narrow.* Since global communication penalties are incurred whenever dataflow crosses cluster boundaries, it must be possible to collocate critical dataflow at a single cluster and to do so without incurring resource contention stalls in the process. That is, critical dataflow must be narrow enough to execute at full speed at a single, small cluster.

2. *Non-critical dataflow must have slack.* In the process of ensuring critical instructions are not delayed, non-critical instructions must not themselves become so much delayed that they are exposed as critical. That is, non-critical instructions must have sufficient slack to shoulder the penalties imposed on them.

I show in this chapter that dataflow readily lends itself to both these requirements. I use for this purpose an *idealized scheduling infrastructure*, which exploits oracular knowledge of the critical path to judiciously allocate instructions to clusters, aiming always to ensure that the most performance-critical are the least likely to be exposed to execution model constraints. This result has two important ramifications. First, distributing execution among a number of out-of-order PEs does not, of itself, fundamentally constrain performance potential. That is, the distributed dataflow-oriented execution model is, in principle, able to match the performance of its monolithic counterpart. Second, attention to the critical path is a viable means for orchestrating distributed execution. Chapters 7 through 9 are devoted to the latter issue, but in a practical setting in which all the idealized aspects of this chapter's analysis must be forfeited.

6.1 Idealized list scheduling

The instruction scheduling infrastructure is depicted in Figure 6.1. The traces I examine are obtained from the timestamp-annotated instruction stream retiring from the back-end of my

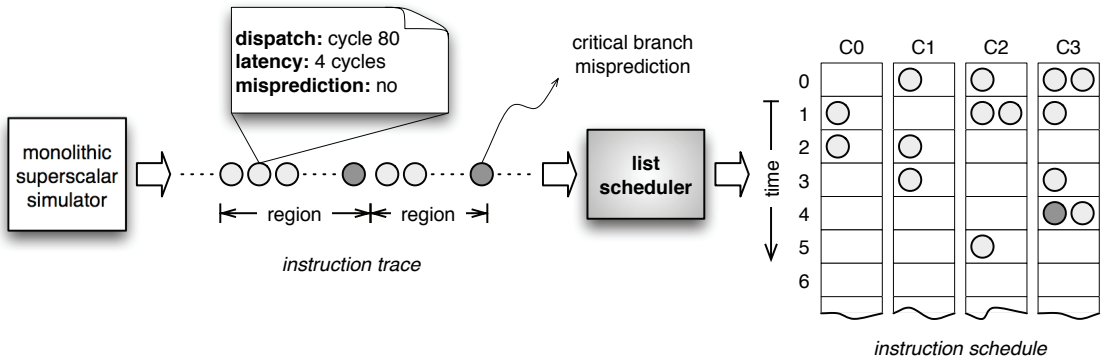


Figure 6.1. Idealized instruction scheduling framework. A timestamp-annotated instruction trace retiring from a simulator for a monolithic machine is partitioned into *regions*, each separated by a critical branch misprediction. A list scheduler processes the trace one region at a time, arranging instructions into a schedule — a placement (at a cluster) and a slotting (in time) of each instruction in the region. In this case, the scheduler is targeting a 4x2w clustered machine. The spans of schedules thus obtained are then summed to yield a total span (runtime) for the whole trace.

simulator. That simulator is configured for a monolithic microarchitecture, so the timing information does not reflect any of the constraints imposed by a distributed execution model. In fact, the only information I glean from the trace, over and above dynamic dataflow dependences, is each instruction's window entry (dispatch) time, its execution latency (cache misses appear as long-latency loads) and, in the case of control flow instructions, whether a branch misprediction was incurred.

To render traces of a manageable size, I use a technique analogous to *interval analysis* [30,48,65]. This involves partitioning the instruction stream into *regions*, each separated by a critical branch misprediction. I deem a misprediction critical if all instructions earlier in fetch order are finished executing by the time the misprediction is resolved, which occurs when correct-path instructions start to enter the machine's window again. I call the time from entry into the window of the first instruction in a region to the resolution of the misprediction at its end the *span* of that region. By definition, all instructions within a region will complete executing before any instructions in the adjacent region start executing, so region spans can simply be summed to yield precisely the net span of the whole trace.

Each region thus formed is supplied to a list scheduler [58], which arranges instruc-

tions into a *schedule* for the target microarchitecture being studied. By schedule I mean a *placement* (at a cluster) and a *slotting* (into an issue slot) of each instruction in the region. The task of the scheduler in so doing is to find an allocation of instructions to issue slots such that the span of the resulting schedule is minimized. It must, in the process, adhere to a number of constraints.

1. *Fetch.* Instructions cannot be slotted before they have entered the window. As Figure 6.1 shows, instructions in the trace are annotated with their window entry (dispatch) time.

2. *Dataflow.* Instructions cannot be slotted until the producers of all of their operands have been slotted and their corresponding execution latencies have elapsed. Again, dataflow dependences (including those via memory) are discerned from the instruction trace.

3. *Issue.* At each time step in the scheduling process, there is a restriction on the number of each type of instruction that can be slotted at each of the target clusters. There is also a limit on the total number of instructions, regardless of type, that can be slotted at each cluster. Restrictions on issue bandwidth imposed by the target machine are therefore properly adhered to.

4. *Communication*. Any values communicated across clusters incur a global communication penalty.

To quantify the quality of the schedules I obtain, I compare their aggregate span to that of the schedules produced when the scheduler is targeted to a monolithic machine (effectively, a single, wide-issue cluster). These monolithic schedules are very much the same as those induced by the issue logic in the monolithic machine from which the instruction trace derives. In fact, the aggregate spans of schedules for the monolithic machine configuration are consistently within 1% of the runtime reported by the simulator for the same

trace — evidence that the monolithic machine is very closely approximating the ideal of a dataflow-oriented execution model.

Earlier, I referred to this scheduling framework as *idealized*. I did so for a number of reasons.

1. *Upstream knowledge.* The scheduler has precise knowledge of the future because it sees all instructions in a region at once. This permits it to very accurately identify, and so give priority to, instructions from which long dataflow chains emanate or which reside on the backward dataflow slice of the region's terminating branch misprediction. By always attempting to collocate consumers with their producers, with precedence being given to those consumers least able to tolerate any penalties, the scheduler tends to keep long, critical dataflow chains collocated and free from resource contention; only the least critical chains incur those penalties.

2. *Downstream knowledge.* Each placement decision made by the scheduler is based on a precise view of the current state of execution induced by all of its prior decisions. This is ultimately because the scheduler performs steering and scheduling at the same time. These two steps are necessarily decoupled in a realistic setting, meaning steering decisions will usually have to be made with minimal, or inaccurate, knowledge of the state of execution at each cluster.

3. *Out-of-order placement.* As a result of its downstream knowledge, the scheduler commits an instruction to an issue slot (and hence PE) only when that instruction is both dispatch- and dataflow-ready. It therefore performs instruction steering out-of-order, implicitly buffering fetched instructions until they are ready to be placed. In effect, the scheduler presumes a monolithic issue queue and treats only the functional units themselves as distributed.

All of these capabilities would, of course, have to be forfeited in a realistic machine. They are idealized here because I am interested in the basic capabilities of the underlying hard-

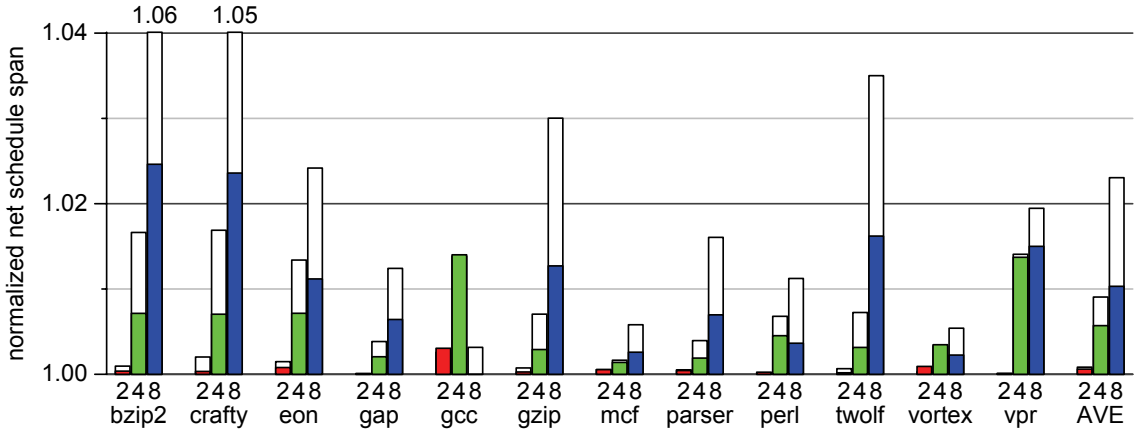


Figure 6.2. Idealized list scheduling. Bars labeled '2', '4' and '8' denote the 2x4w, 4x2w and 8x1w clustered configurations, respectively. Each bar plots the aggregated span (the time from first instruction slotting to the completion of the last instruction) of schedules produced for each region in 100-million instruction traces. Bars are normalized to the aggregated schedules produced for the 1x8w machine. Those baseline schedule spans are consistently within 1% of the simulated runtime for the monolithic machine from which the traces derive. In all configurations, the shaded portion of the bar corresponds to a 2-cycle global communication penalty; the white portion shows the effect of doubling that penalty to 4 cycles.

ware, not in the penalties introduced by steering and scheduling policies needed for its management.

6.2 Results

Figure 6.2 plots the schedule times I obtained using the above approach. Two points warrant mention. First, with a 2-cycle global communication penalty, all of the clustered machine configurations achieve average performance that is no more than 1% slower than the 1x8w configuration. Moreover, increasing the penalty to 4 cycles has only a modest impact on the average, with the 2x4w and 4x2w machines still below a 1% penalty, and the 8x1w now marginally above 2%. Clearly, partitioning the underlying hardware does not, of itself, impose any fundamental limits on how close a clustered machine's IPC can approach that of its monolithic equivalent. This means that state of the art schemes, which do not come close to these very small performance losses, are significantly underperforming; the hardware is capable of delivering much better performance.

A second observation relates to benchmarks `bzip2`, `crafty` and `vpr`, whose performance, particularly on the 8x1w configuration, stands in contrast to the other benchmarks. By means of critical path analysis, I find that the bulk of the discrepancies — about 70% of the extra cycles in the benchmarks overall, and over 80% in the three worst-performing benchmarks — can be attributed to a single cause: *convergent dataflow*. Arising wherever there are dyadic instructions, this is, in general, a common phenomenon; it is significant, however, only when the dyadic instruction is on the critical path *and* both its incoming dataflow edges have little or no slack — a comparatively rare phenomenon. Convergence in such cases constitutes a fundamental problem for clustered machines because it demands an ability to exploit ILP without delay. When clusters are narrow, this requirement translates into unavoidable performance penalties: collocating the converging chains will inject contention stalls onto the critical path because the critical ILP will be serialized; and attempting to exploit that ILP by distributing it across clusters will inject communication penalties onto the critical path.

Figure 6.3 illustrates the problem with an example from `bzip2`. In this case, convergence manifests itself as global communication penalties (hence the large performance difference between a 2- and a 4-cycle penalty on that benchmark). On the 8x1w configuration, the best possible performance is obtained by mapping the converging chains to distinct PEs, as shown in Figure 6.3(b). There is a necessary 2-cycle global communication penalty incurred in the process, but this is better than the contention stalls that would have been incurred had those chains been collocated. Figures 6.3(c) and 6.3(d) show that the 4x2w and 2x4w machines prefer collocating the converging chains, since their improved local issue bandwidth renders contention stalls less severe (they are entirely eliminated in the 2x4w machine). Even so, achieving those schedules is very difficult because it demands advance knowledge that convergence is imminent: the collocation cannot be guided by dataflow relationships because the two chains are independent.

Convergence in `vpr` incurs contention stalls. Figure 6.4 shows that this occurs in a

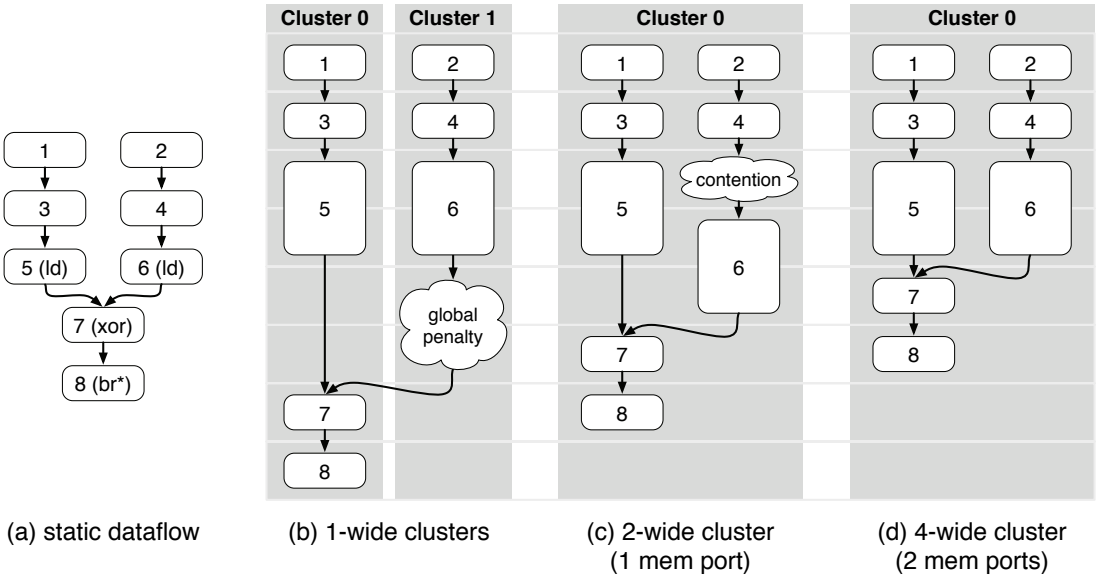


Figure 6.3. An example of convergent dataflow in `bzip2`. (a) Dataflow leading into a critical branch misprediction. Nodes are labeled in their fetch order and their operation type (unlabeled instructions are single-cycle integer operations). (b) The optimal allocation for 1-wide clusters incurs one global communication penalty; there would be 3 cycles of contention if it was assigned to one cluster. (c) With 2-wide clusters, each with a single memory port, there is a single cycle of contention for the memory port. (d) With clusters that each have 2 memory ports, this code can execute at full speed.

dataflow "hammock", where a single instruction produces a value for two chains of consumers (diverging dataflow), which, in turn, subsequently converge at a dyadic consumer. When this occurs, the scheduler attempts to collocate both consuming chains with the single producer. On both the 8x1w and the 4x2w configurations, this incurs a partial serialization of the two independent chains of load instructions, which manifests itself as resource contention. Hence the very similar performance results achieved by these two configurations on `vpr` (recall Figure 6.2). On the 2x4w machine, however, there is enough local memory issue bandwidth to simultaneously issue both chains of loads, permitting this code to issue at the peak rate allowed by the dataflow.

Note that the slowdowns experienced by the 8x1w and 4x2w configurations on this particular code sample are *unavoidable*. Had the steering logic attempted to avoid contention stalls by steering each of the load chains to a distinct cluster, it would merely have substituted global communication penalties for contention stalls. Simply put, this dataflow

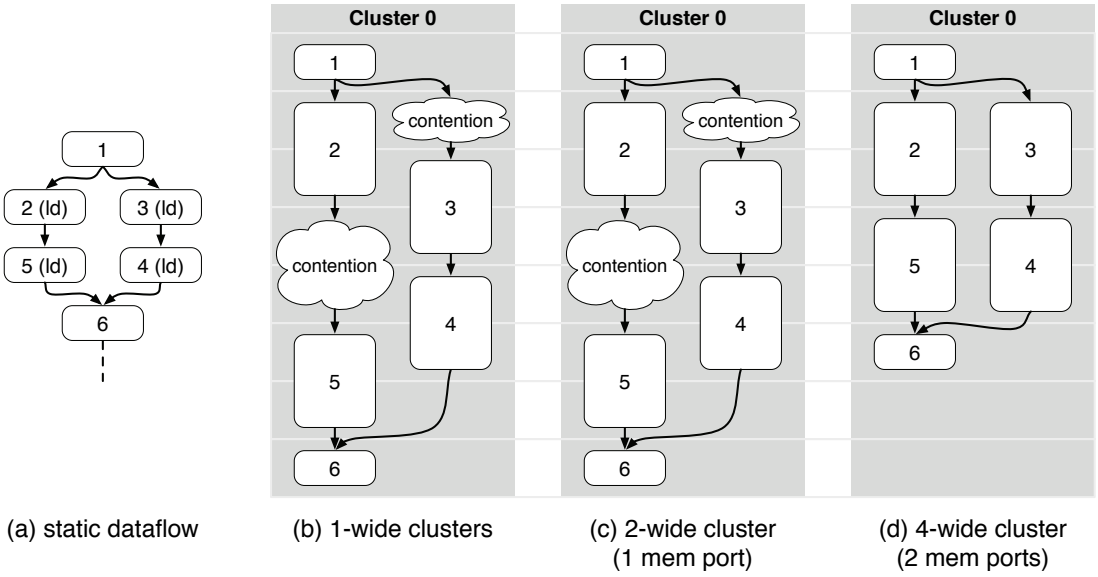


Figure 6.4. An example of convergent dataflow in `vpr`. (a) Critical dataflow leading into a mispredicted branch (branch not shown). As per Figure 6.3, nodes are labeled in their fetch order. (b), (c) The 8x1w and 4x2w configurations, both being able to issue only one memory operation per cycle, introduce 2 contention stalls into the critical dataflow hammock. (d) The 2x4w configuration, in which each cluster has 2 memory ports, can execute this code at full speed.

embeds more ILP than a single cluster is able to extract; and, because all of it is critical, no subset of its dataflow edges can tolerate contention stalls or communication penalties.

In general, convergent dataflow poses a difficult problem for clustered machines because dealing with it requires advance knowledge that convergence is imminent. Even with forward knowledge, which the idealized scheduler has, there is no single policy that caters for all scenarios. There are cases where collocation is better (*e.g.* `bzip2`-like convergence on the 4x2w configurations) and cases where load-balancing is potentially better (*e.g.* large hammocks on the 8x1w configurations). For the 8x1w configuration, in particular, the problem is a fundamental one because the demand for parallelism necessarily incurs either global communication penalties or contention stalls. That said, the overall effect is still small: even the 8x1w configuration is never more than 3% slower than the monolithic schedule (for 2-cycle global communication). Clearly, critical dataflow in these benchmarks seldom demands support for intra-cluster parallelism.

6.3 Summary

Prior research into clustered microarchitectures has consistently reported substantial IPC losses relative to equivalent monolithic designs, leading ultimately to the view that performance penalties are unavoidable. The results I presented in this chapter counter that trend. For the family of clustered machines I studied here, I consistently find that *there exist* instruction placements that yield performance very close to that of equivalent monolithic designs. It must be concluded, therefore, that critical dataflow in the SPEC Integer programs is narrow enough to be collocated at even 1-wide clusters. Equally, non-critical dataflow is very tolerant of communication and contention penalties — if those penalties are judiciously imposed. In short, if the right dataflow chains can be accurately identified and appropriately given preferential treatment, then a distributed dataflow-oriented execution model is, in principle, no different from a monolithic one.

Chapter 7

Overcoming execution constraints[†]

One important implication of the last chapter's idealized scheduling results is that previously published clustered machine studies, which consistently report substantial IPC losses relative to a resource-equivalent monolithic machine, must attribute their performance problems to specific implementation choices, not to inherent limitations imposed by the underlying execution model. That inherently good performance potential has now been demonstrated, and that instruction criticality was the means by which this was done, both justifies and motivates seeking an effective critical path-aware scheme for exploiting the hardware's potential to the fullest. This is the challenge I now take up.

I will not yet propose specific implementation *mechanisms*, but rather focus here on *policies* for overcoming execution model constraints. I start by examining the *focused steering and scheduling* infrastructure developed by Fields *et al.* [35], which constitutes the state-of-the-art in terms of using criticality to manage a clustered machine's execution resources. In Section 7.1, I use critical path analysis to show that, although it was designed specifically to keep critical instructions free of global communication and resource contention, their scheme frequently injects both penalties onto the critical path. Specifically, I find that known-critical instructions suffer a number of resource contention stalls, in spite of a dynamic scheduler that gives priority to such instructions. I also find that global communication is incurred by known-critical instructions, in spite of dynamic steering logic aiming always to collocate instructions with their critical producers.

In Sections 7.2 through 7.4, I then explore the underlying causes for these problems.

[†] The content of this chapter derives from work published at the 38th International Symposium on Microarchitecture [84]

I show in Section 7.2 that the criticality predictor used by Fields *et al.* — or, indeed, any counter-based predictor trained by postmortem analysis of the critical path — is inherently limited. There are two principal reasons for this. First, predictor inaccuracies, which are inevitable, will lead to cases in which it becomes impossible to correctly prioritize between instructions that frequently appear on the critical path. Second, even a perfectly accurate predictor solves only part of the problem: what is needed is not so much an ability correctly identify the critical path, but rather an ability to correctly prioritize among *all* in-flight instructions, be they critical or not. To circumvent both these problems, I introduce a new criticality metric, *Likelihood of Criticality (LoC)*, which explicitly quantifies the frequency at which each static instruction is critical.

With LoC in hand, it is then possible to develop three policies for removing global communication and contention stalls from the critical path.

1. *LoC-based scheduling.* By tagging each dynamic instruction with a predicted LoC value, the dynamic instruction scheduler can prioritize among critical instructions more intelligently than can a scheduler using binary criticality (Section 7.2).

2. *Stall versus steer.* In execute-critical regions of code, it is better to stall steering when a cluster is full than it is to steer critical instructions away from their producers (Section 7.3). The LoC metric can be used to guide this selective stalling behavior.

3. *Proactive load-balancing.* On machines with narrow clusters, load-balancing (sending consumers away from their producing instructions) should be performed so as to steer all but the most critical consumer to other clusters (Section 7.4). Because the most critical consumer is seldom the first one to appear in fetch order, this requires learning which early consumers should be proactively steered away to make room for a subsequent, more critical, consumer.

For each of these policies, I provide code examples to give the intuition behind why they are necessary. Then, in Section 7.5, I complement those anecdotal examples with performance

results to demonstrate the efficacy of the policies described above. I show reductions in the penalty due to clustering of one-half to two-thirds for 2-, 4-, and 8-cluster machines, which brings the slowdowns relative to a monolithic machine to 2%, 4% and 6%, respectively.

7.1 The state of the art

In this section, I show that the focused steering and scheduling policies developed by Fields *et al.* [35], which constitute the state of the art in clustered machines, exhibit slowdowns that are about an order of magnitude worse than those achieved with the idealized list scheduling study presented in the previous chapter. I then use critical path analysis to zoom in on the behavior of those policies, which leads to important insights into why they are not performing well.

7.1.1 Focused steering and scheduling

The focused steering and scheduling scheme developed by Fields *et al.* equips the pipeline with a criticality detector that samples the retiring instruction stream, looking for instructions whose execution resides on the program's critical path. That detector implements the dependence graph model of instruction criticality that I reviewed in Chapter 1 (Figure 1.2, page 30). Results from the detector feed a criticality predictor, which is a PC-indexed table of saturating counters. The counter for a given PC is incremented when the corresponding instruction is detected as critical, and decremented otherwise; when that value exceeds a certain threshold, the instruction is classified as critical. The machine uses a dependence-based instruction steering policy [49], modified to use output from the criticality predictor so that whenever there is a choice of cluster to which a consumer can be sent, the one holding the critical producer is given preference. In addition, instruction scheduling logic is modified to give priority to critical instructions, the objective being to reduce the exposure of critical instructions to contention-induced stalls.

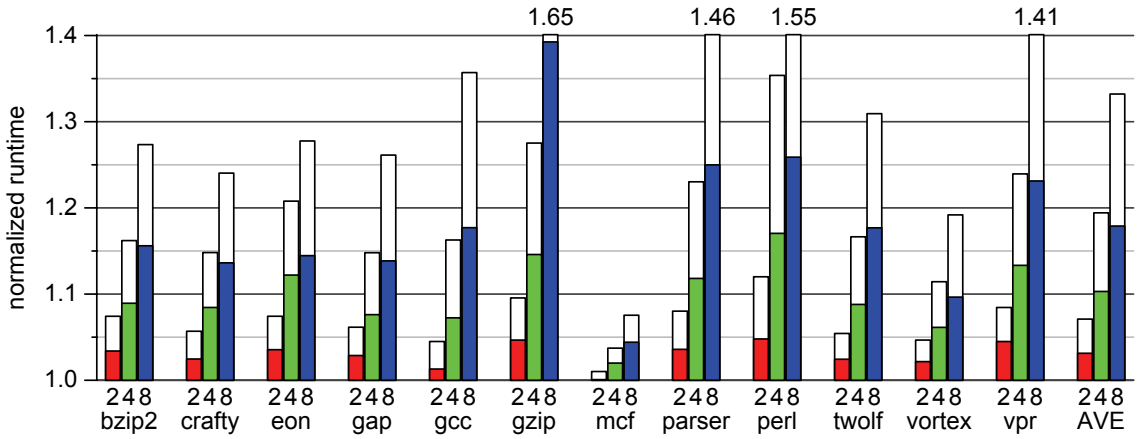


Figure 7.1. Focused steering and scheduling. Like Figure 6.2, bars labeled 2, 4 and 8 denote the 2x4w, 4x2w and 8x1w machines. In contrast to that figure, however, the scale on the y-axis is now larger by an order of magnitude. As before, the shaded portion of each bar shows performance with a 2-cycle global communication penalty; the unshaded portion, a 4-cycle penalty.

I incorporated the Fields critical-path detection logic into my simulation environment and implemented the focused steering and scheduling policies. The performance results I obtain, which are in general agreement with those published by Fields *et al.*, are plotted in Figure 7.1. When a 2-cycle global communication penalty is modeled, performance of the 2x4w configuration is always within 10% of the monolithic machine, but the 4x2w configuration is frequently in excess of 10% and, in the case of `perl`, close to 20%; the 8x1w configuration averages a slowdown of almost 20%. Increasing the communication penalty to 4 cycles generally doubles the performance losses. Overall, these figures correspond to at least an order of magnitude increase in the penalties incurred by the list scheduler in the previous chapter (Figure 6.2).

7.1.2 Analysis of the lost cycles

The above data gives rise to the obvious question as to why the focused policies, though designed specifically to keep critical instructions free of communication penalties and contention stalls, are failing to exploit the performance the hardware is capable of delivering. Obviously global communication and resource contention are ultimately to blame, but ag-

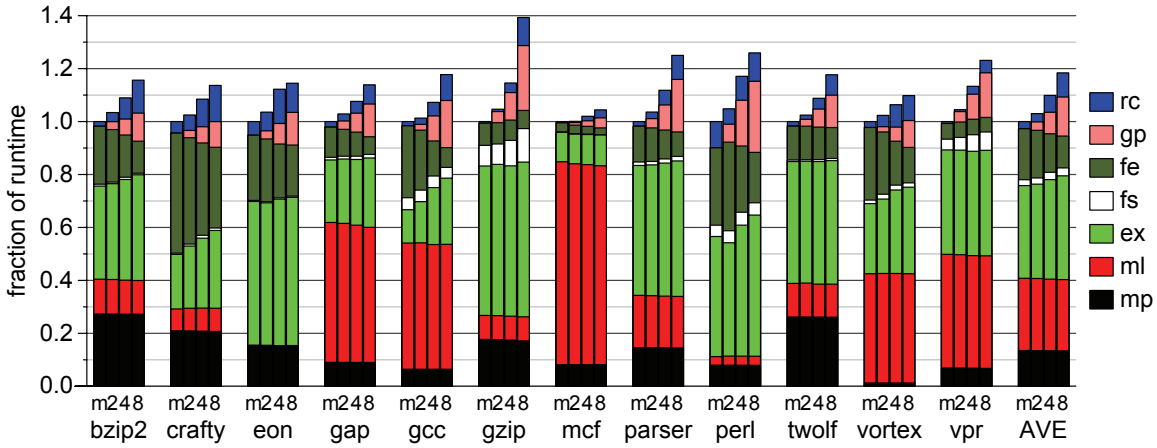


Figure 7.2. Critical path signature. Bars labeled 'm', '2', '4' and '8' show, respectively, the critical path breakdown for the monolithic, 2x4w, 4x2w and 8x1w machines. All bars are normalized to the leftmost one in each group (*i.e.* the monolithic machine). The critical path categories ('rc', 'gp', *etc.*) are described in Table 1.1 (page 32). The global communication penalty is fixed at 2 cycles.

gregate statistics relating to these metrics are not meaningful: many instructions can incur these penalties without impacting the overall execution time [33]. Only those cases in which global communication and contention stalls actually contribute to runtime are of interest. To attribute cycles to just those cases, I make use of critical path analysis, just as I did in Chapter 3 to diagnose problems with dependence-based scheduling. To briefly reiterate, this involves post-processing an execution trace to delineate the critical path (as per the dependence graph model from Fields *et al.*). I then attribute cycles to various aspects of performance — the categories enumerated in Table 1.1 (page 32). I should emphasize that these attributions are not always unique: programs frequently exhibit parallel and near-critical paths [34]. As a result, a performance improvement is not always guaranteed if slowdowns on only one critical path are addressed. Nevertheless, this methodology is accurate enough to provide insight into why the focused policies are not working as well as they could.

Figure 7.2 shows the composition of the critical path for the focused steering and scheduling policy (same data as Figure 7.1). The cycle breakdown shows that, in spite of the focused policies, the critical path frequently suffers from distributed execution model penalties: it often crosses clusters (the 'gp' category) and it often stalls because critical

instructions are not being executed as soon as they are ready (the 'rc' category). On occasion, the execution time attributed to these two effects exceeds the slowdown relative to the monolithic machine. This does not suggest that the clustered machine will outperform the monolithic one if those stalls are eliminated. Rather, this is indicative of a shift in the critical path. By introducing stalls, the clustered core has forced what was previously a near-critical path to become critical.

Much of the observed change in the critical path is a shift from fetch- to execute-criticality.[1] This means the back end is no longer keeping up with the front end, causing the window to fill quicker than it drains. Instructions are, as a result, more likely to be put into the window before they are data ready, making them execute-critical. While this is evidence that the partitioned core is having an impact, it does not provide insight into why clustering-induced stalls remain, despite the criticality-based steering and scheduling policies.

To answer that question, I can zoom in on the critical path to find the main contributors to contention and communication penalties. Figure 7.3 summarizes the results of that analysis. Figure 7.3(a) shows that as much as two-thirds of resource contention stalls come from instructions delayed in spite of being correctly predicted as critical. That is, the contention does *not* arise because the criticality predictor produces false negatives: instructions that are truly critical are correctly predicted as such. I will show in the next section that the problem is ultimately caused by false positives, since it is multiple predicted-critical instructions that are contending for the same resources. Figure 7.3(b) shows that the dominant cause for global communication penalties is load-balance steering. This occurs when an instruction's critical source operand will be produced by a cluster that is currently full. The consumer is in such cases assigned to the cluster with the fewest in-flight instructions.

In the sections that follow, I demonstrate these effects by way of code examples and then discuss changes in policies that can address them. First, in Section 7.2, I show that

[1] I explained the notions of fetch- and execute-criticality in Chapter 1, Section 1.3.3 (see Figure 1.3, page 33).

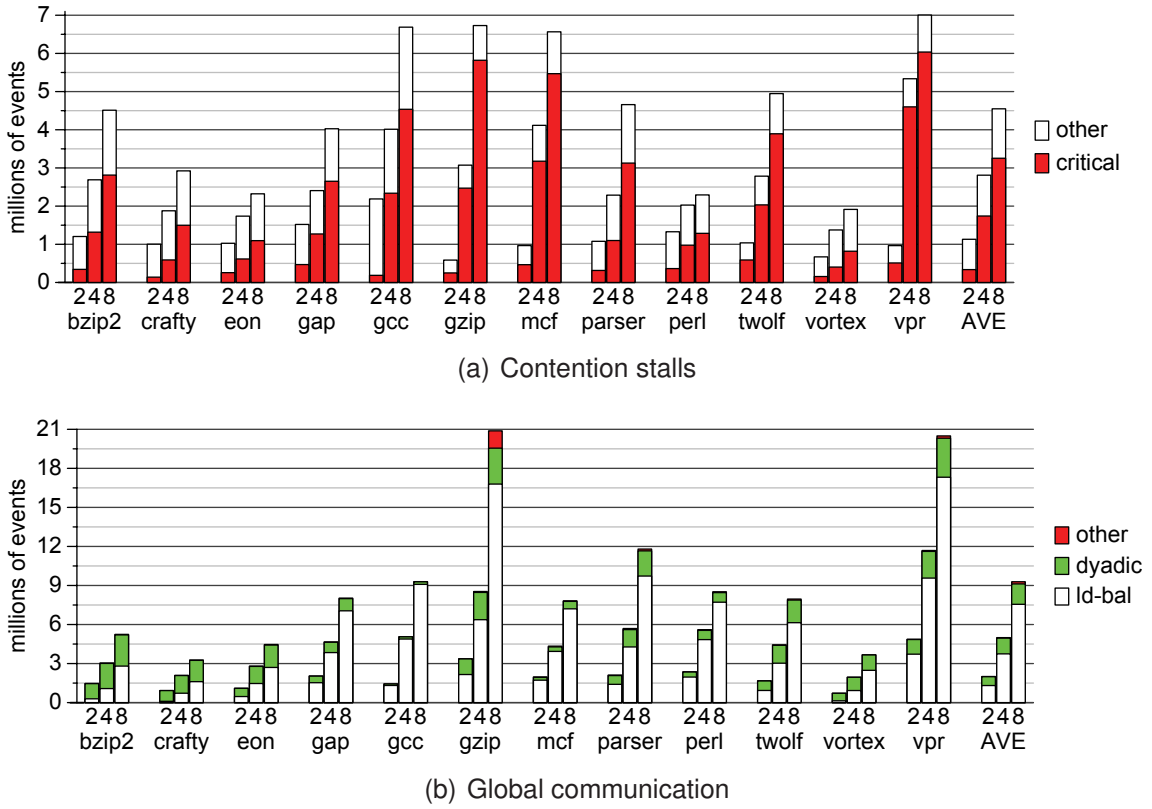


Figure 7.3. Where the lost cycles went. (a) Contention stalls among critical instructions are incurred predominantly by instructions that have been predicted critical. (b) Global communication is incurred by critical instructions mainly as a result of load-balance steering. Only in `bzip2` and `crafty`, which have an abundance of convergent dataflow, do dyadics dominate.

the contention problem arises from Fields's binary notion of criticality. What is needed is a more continuous spectrum that facilitates distinguishing between instructions that are often on the critical path. I introduce the notion of *Likelihood of Criticality* for this purpose. In Section 7.3, I then demonstrate that load balancing is exactly the wrong thing to do when code is execute-critical; it is preferable to *stall* instruction steering when the desired cluster is full. I also show that likelihood of criticality is a good metric for discerning execute-critical regions of code, and hence for driving the decision to stall rather than steer. Finally, I show in Section 7.4 that a positive side effect of stalling on critical instructions is an improvement in the distribution of ready instructions. To amplify those benefits, I introduce a *proactive load-balancing* scheme to push non-critical consumers away from critical producers.

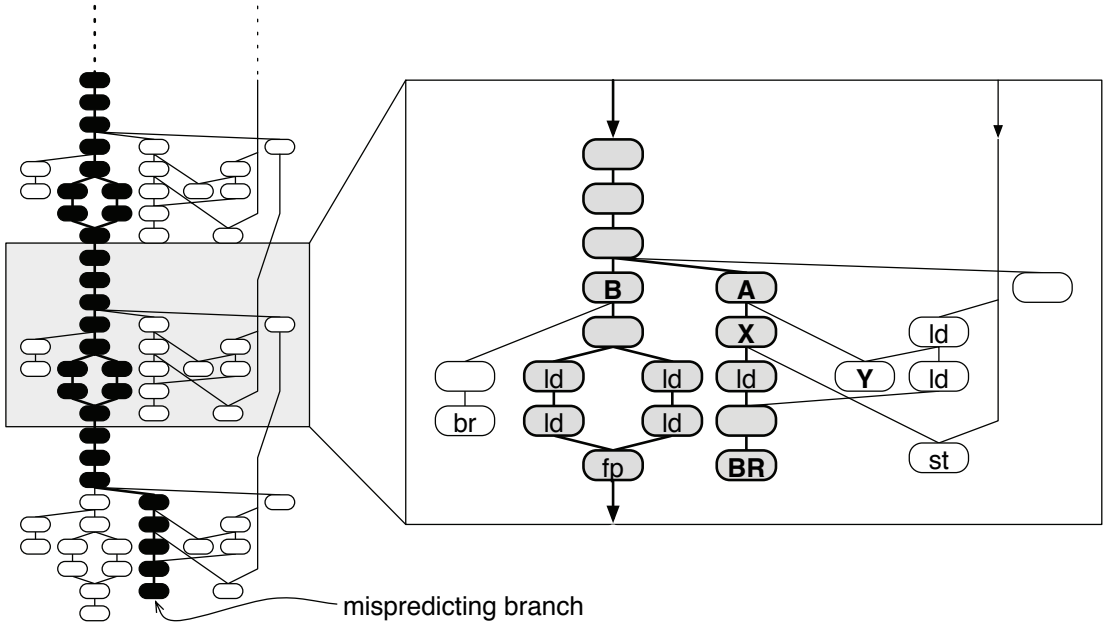


Figure 7.4. A major source of contention-related stalls. The critical path, which ends in a mispredicted branch (BR), is highlighted in the diagram. Both instruction A (on the rib) and B (on the spine) are predicted critical, but instructions on the spine are actually critical more often.

7.2 Likelihood of criticality

The effect that underlies the contention stalls incurred by predicted-critical instructions is most easily demonstrated with an example. Figure 7.4 shows a loop from the benchmark vpr. The dataflow exhibits the same *spine and ribs* structure that I described much earlier in Chapter 3 (see Figure 3.7, page 57, for example). The dominant spine, which flows through the instruction labeled B, computes a loop-carried dependence. Dataflow periodically diverges from this spine to feed the ribs, which terminate on stores and branches; the rib that starts with the instruction labeled A includes a hard-to-predict branch (BR).

A contention problem arises between instructions A and B. Both of them frequently appear on the critical path because they are both on the backward slice of the critical branch misprediction. They are therefore both classified as critical by the criticality predictor. And since both instructions consume from the same source register, they are routed by the dependence-based steering policy to the same cluster. Further, being ready to execute at

the same time, they will contend for an issue slot on a machine with 1-wide clusters. The scheduler, because it sees both as critical, breaks ties by choosing the older one — in this case, instruction A. This is the wrong choice for every iteration but the last, because only in the last iteration is instruction A actually critical; on all other iterations, instruction B, the truly critical one, incurs a contention stall.

One way to tackle this problem is to try to improve the criticality predictor so that it "tunes into" the changing behavior of instruction A, analogous to how a local branch predictor can track the taken/not taken behavior of a given branch. However, since instruction A is critical precisely when the branch BR is mispredicted, this kind of adaptability on the part of a criticality predictor would amount to knowing exactly when the branch will be mispredicted. In general, achieving very accurate criticality predictions is fundamentally hard — in the same sense that very accurate control flow and cache hit/miss prediction is fundamentally hard.

I want to stress also that this contention problem is not merely an unfortunate interaction between a particular counter-based prediction technique and the specific rate at which instruction A is critical. It is true, for example, that choosing a higher counter threshold value in the predictor might cause A to no longer be predicted critical, yet would leave B as always predicted critical. But such a change, though it would solve this particular problem, would no doubt have an adverse effect on other critical/not-critical distinctions. In fact, no single threshold value can work in general. This is because what is ultimately needed is an ability to distinguish between a very wide range of instructions, not just between those that are frequently critical. Returning to Figure 7.4, for example, there is also a need to distinguish between the two consumers of instruction A's value, namely instructions X and Y. Like A and B, these two instructions exhibit different criticality behavior relative to one another, but their differences will not look anything like those between A and B. What is lacking here is an ability to gauge the *relative* importance of any two instructions, not merely an ability to distinguish the critical path from the rest.

These problems are inevitable if the prediction framework is confined to classifying instructions as either critical or not critical. In short, *binary* criticality predictions are too coarse grained a means for aggregating past criticality behavior; too many different types of dynamic behavior are thereby rendered indistinguishable. A more useful metric would reflect the *frequency* at which each instruction is critical. This would permit instructions `A` and `B` to be easily distinguished — `B` is critical much more often than is `A`; likewise for instructions `X` and `Y`. To this end, I introduce the *Likelihood of Criticality (LoC)* metric. I assign an LoC of $n\%$ to a static instruction if $n\%$ of all of its previous dynamic instances have been critical. That is, an instruction's LoC is the frequency at which it has been critical in the past.

LoC can be viewed as a measure of the *expected cost* of making a wrong decision on an instruction. For example, imposing a 2-cycle global communication delay on an instruction with an LoC of 75% would, for each instance of that instruction, add about 1.5 cycles to the program's execution time. Likewise, an instruction with an LoC of 25% would incur 0.5 cycles for each instance. Whereas these two instructions would both be classified as critical by a typical binary predictor, being able to distinguish them via their very different LoC values, and hence give priority to the former over the latter, yields an expected savings of 1 cycle on each of their instances. Figure 7.5 shows that the LoC metric exhibits a wide distribution of values, indicating that it does indeed have the potential to distinguish numerous degrees of criticality. The vertical dashed line in the figure approximates the granularity achieved by the binary predictor proposed by Fields *et al.* [35].[2]

An important property of LoC is that it tends to be a stable property. That is, an instruction's *tendency* to be critical in the past is a very good indicator of its *tendency* to be critical in the future. Though I will discuss this issue at length in Chapter 8, I will state at this point that modifying the idealized list scheduler (from Chapter 6) so that it prioritizes instructions

[2]The Fields predictor uses a 6-bit saturating counter that increments by 8 when training critical, and decrements by 1 when training non-critical; the threshold value for predicting critical is 8. Thus, 1 in 9 instances being critical is sufficient for an instruction to be classified as critical.

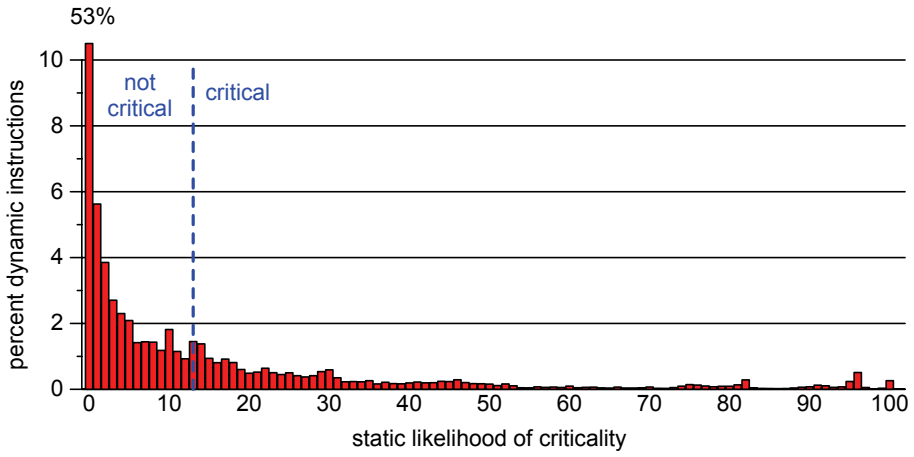


Figure 7.5. Distribution of LoC values. Data is averaged across all 12 benchmarks.

based solely on LoC values continues to yield very good results. That is, replacing the list scheduler's idealized knowledge of per-instance criticality with knowledge only of average previous criticality has only a marginal impact on the quality of the resulting schedules: with a 2-cycle global communication penalty, the average performance loss remains below 3% for all clustered machine configurations.

Another important property of the LoC metric is that it succinctly expresses dynamic behavior in terms of a single, numerical property of each static instruction. By contrast, a metric like slack [33] is much harder to express as a static property. This is because slack is measured as a cycle count for each dynamic instance, so different instances can have very different slack values. For example, branches, when mispredicted, have no slack; when predicted correctly their slack is very large, limited only by the size of the instruction window. From a static instruction point of view, this variation can be expressed as a histogram of the different amounts of slack observed by an instruction, but comparing two instructions on that basis is not very practical.

I want to emphasize again that LoC does not improve the ability to accurately predict the true criticality of any given dynamic instruction instance. Rather, it permits choosing, with a high probability of being correct, which of two dynamic instances ought to be given preferential treatment. This does not require knowing whether either of those instances is

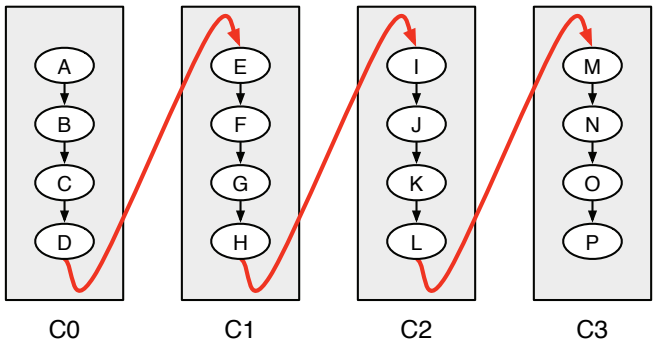


Figure 7.6. The problem with load-balance steering. A load-balancing policy causes a single dependence chain to be spread across all of the clusters. Because that chain is execute-critical, the global communication penalties thereby incurred contribute directly to runtime. In this case, performance would be better served if steering had instead stalled when the desired cluster was full.

actually critical. Thus, LoC improves the ability to accurately gauge *relative* criticality, and hence to more judiciously manage resource allocation. In Section 7.5, I show that this capability can be used to reduce the number of contention-related stalls by a factor of two. Perhaps more importantly, LoC can serve as a useful metric for controlling the two resource allocation schemes I describe in Sections 7.3 and 7.4.

7.3 Stall versus steer

I showed in Section 7.1 that the dominant source of global communication on the critical path is load-balance steering. In this section, I show how this problem can arise in regions of code that are execute-critical.

To understand the underlying problem, consider the hypothetical program shown in Figure 7.6. The code, which consists entirely of a single chain of dependent, unit-latency add instructions, has an ILP of 1 and experiences no branch mispredictions. It can therefore be fetched much faster than it can be executed. The program is thus *execute-critical*; its fetch rate does not affect its execution rate. On a clustered machine using a dependence-based steering scheme, this dependence chain will quickly fill the issue queue at the cluster to which it is steered (it is fetched faster than it can execute). When this happens, the

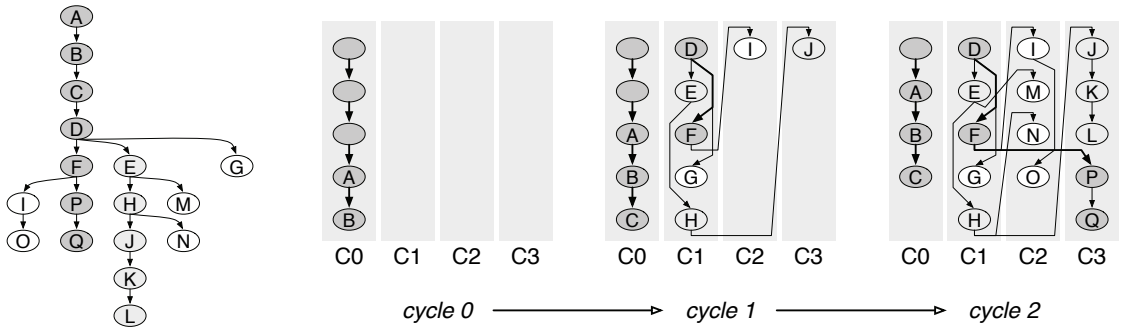


Figure 7.7. Spreading the critical path across clusters. The dataflow graph on the left is distilled from the earlier `vpr` example (Figure 7.4). Instructions are labeled in their fetch order and are shaded according to their relative criticality (darker is more critical). Instruction `L` is a frequently mispredicted branch. The three timing diagrams to the right show how plain dependence-based steering will distribute those instructions among 4 clusters. In cycle 0, cluster `C0`, which holds the critical spine, fills up. In cycle 1, one instruction of the spine executes, making room for instruction `C`, which fills the cluster. As a result, instruction `D` is load-balanced to cluster `C1`; instructions `E` through `H` follow it there. Because the second cluster is now full, instructions `I` and `J` are load-balanced to different clusters. In cycle 2, instruction `P` cannot be steered to its producer's cluster (`C1`), causing it also to be load-balanced to cluster `C3`. In this example, the critical path `ABCDFPQ` is lengthened by two unnecessary global communication penalties.

existing dependence-based steering policy reacts by load-balancing: it assigns the next instruction in the chain to the least-utilized cluster. And so the process repeats: upon filling the second cluster, load-balancing redirects the chain to a third cluster until it too fills. As Figure 7.6 shows, the net effect of this load-balancing is the introduction of one global communication delay into the dependence chain every N instructions, where N is the size of a cluster's issue queue.

For this hypothetical piece of code, it would be preferable to *stall* the steering logic until a space becomes available at the desired cluster. Doing so would not slow the program down because, as I noted already, its fetch rate is not determining its execution rate. Instead, stalling would eliminate the global communication penalty from the dataflow chain entirely, permitting the clustered machine to match the performance of a monolithic one.

Real programs are not so simple, of course, but this type of behavior does indeed occur. Figure 7.7 is an example of how the critical path in the earlier `vpr` code sample (now slightly simplified) incurs even more communication penalties than the single dependence chain. This is likely the effect that is responsible for the observation, made by Balasubra-

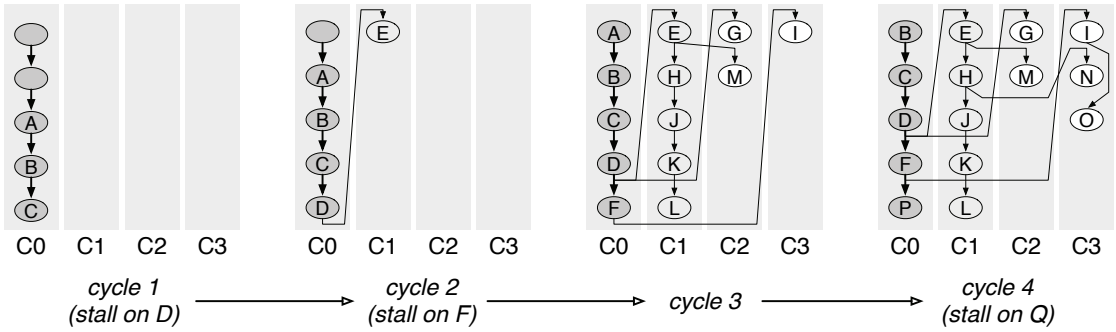


Figure 7.8. Proactive stalling. The timing diagrams show the effects of selective stalling when execute-critical instructions cannot be sent to their producer's cluster. Starting from the cycle 0 state in Figure 7.7, the leftmost diagram (for cycle 1) shows only instruction `C` being steered, because `D` has high LoC and its desired cluster is full. In cycle 2, there is space for `D` and, because `E` is not sufficiently critical, it is load-balanced to cluster `C1`. Instruction `F` manages to collocate with its producer in cycle 3, after which non-critical instructions `G` and `I` are load balanced; instructions `H`, `J`, `K` and `L` are then dependence-based steered to cluster `C2`. In cycle 3, instructions up to `P` are steered, but `Q` must wait until the next cycle. This approach does not fetch the program as quickly as before, but it keeps the critical slice collocated at one cluster; and there is no benefit to fetching the program faster than it can execute.

monian *et al.*, that low ILP programs perform better on 4 1-wide clusters than on 16 1-wide clusters [6]. In effect, a smaller number of clusters increases the chance (from 1 in 16 to 1 in 4) that critical dependences are steered to the same cluster when load-balancing occurs.

The potential benefits of stalling the front-end have also been reported by González *et al.* [39]. They propose a scheme that uses the number of in-flight instructions at each cluster as a means for controlling the decision to stall. However, cluster load is a very coarse, and potentially misleading, measure of the phenomena that make stalling beneficial. In the hypothetical code example, stalling makes sense only because the program is in an execute-critical region. However, as I explained much earlier (Section 1.3.3), some program regions are *fetch critical*, in that they need to be dispatched as fast as possible into the window in order to reach a critical computation, such as a mispredicting branch slice, that occurs in the future.[3] When code is fetch-critical, stalling is precisely the wrong thing to do. Rather, steering logic should resort to load-balancing in such regions, thereby ensuring forward progress to the critical computations that lie ahead.

[3]The example code from `twolf`, which I described in Chapter 3 (Section 3.2) is a good example of fetch-critical code. See, in particular, Figure 3.11 on page 62.

The LoC metric can be used to distinguish cases where stalling is preferable to steering. Instructions with a high LoC value are likely to be execute-critical, so stalling is preferable; those with a low LoC value are probably fetch critical, in which case stalling would do more harm than good. Empirically, I have found stalling instructions with an LoC exceeding a 30% threshold strikes a good balance. Figure 7.8 shows how the earlier `vpr` example behaves when instructions with high LoC values (those that are darkly shaded) are stalled and all others are load-balanced.

7.4 Proactive load-balancing

There is a positive side-effect to the stall-versus-steer policy. Returning to Figure 7.8, it is clear that the machine is doing a good job of distributing parallel work among the clusters. For example, instructions `E`, `G`, and `I` are not sufficiently critical for stalling, so they are load-balanced to other clusters where they can execute in parallel with the primary dependence chain. This is in contrast to the non-stalling scheme (Figure 7.7), which sends all of `D`'s successors to the same cluster, where they will contend for a single issue slot.

While this improved distribution of parallel work is indeed desirable, relying exclusively on the selective stalling mechanism to achieve it has a number of drawbacks. First, it only occurs after a cluster has filled: it is *reactive*. Second, it requires that there be a clearly identifiable critical dependence chain, yet some high-ILP regions have no instructions with sufficiently high LoC. Finally, the load-balancing applies only to non-critical instructions that diverge from the main dependence chain; instructions that diverge from other, less-critical chains do not likewise benefit.

The code example in Figure 7.9 demonstrates why a more general, *proactive* load-balancing scheme is needed. When dynamically unrolled, this code forms two trees of diverging dataflow. Because a dependence-based steering mechanism will try to collocate instructions with their dataflow producers, each tree will be assigned to a single cluster until

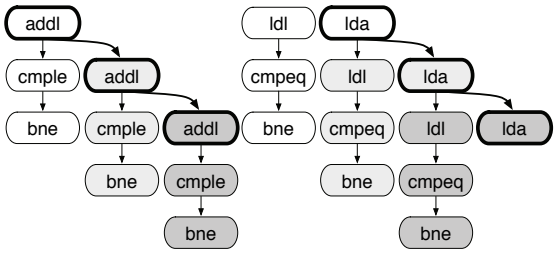
```
int a = ...
int* A = ...

for (i=0; i<N; ++i) {
   if (A[i]== a) {
      break;
   }
}
```

(a)

```
L7: addl  $4, 1  -> $4
    ldl   $7 <- 0($2)
    cmple $4, $5 -> $3
    lda   $2 <- 4($2)
    cmpeq $7, $0 -> $6
    bne   $6, L3
    bne   $3, L7
L3: ...
```

(b)

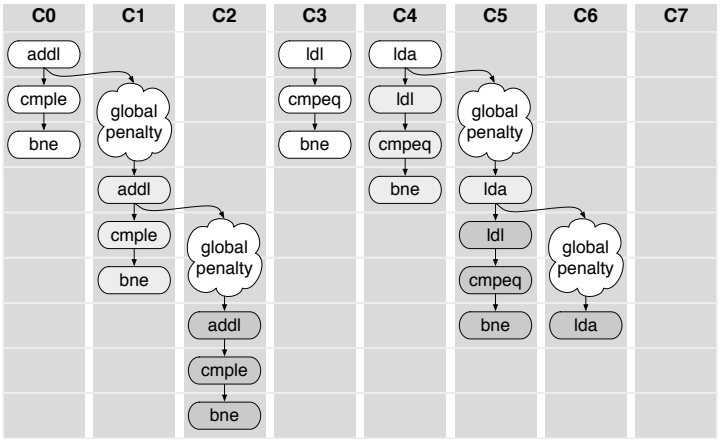


(c)

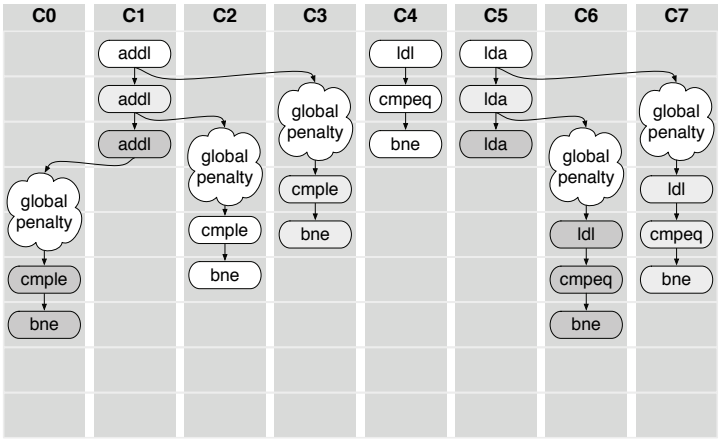
Figure 7.9. An example loop with divergent dataflow. (a) C-code for a loop with an early exit. (b) The corresponding Alpha assembly code, which has been optimized by the compiler to have two separate loop-carried dependences. (c) A dynamic dataflow graph with three iterations of the loop (each shaded in a different color) that shows how the instructions diverge from the loop-carried dependences.

it fills. This is clearly not the desired behavior. For example, on the 8x1w configuration, it will introduce contention stalls that will cause the branches to resolve 5 to 10 instructions later than they would have on a monolithic machine.

One approach that has been adopted for proactive load-balancing is implemented by the dependence-based steering policy for laned machines (Part I), which steers only the first dependent instruction to a given producer, all others being load-balanced [52, 76]. This does a good job of distributing the consumers of an instruction, but it has the drawback that it will introduce global communication onto the critical path if the first consumer (in fetch order) is not the most critical one. The code in Figure 7.9 is an example of precisely this problem. Here, the most critical consumer of the loop-carried dependence produced by the `addl` instruction is the next instance of itself, yet this is also the last consumer of that value (it must be because it performs a destructive update of the register). If the global communication penalty is 2 cycles, steering this critical consumer to a different cluster reduces the IPC of this code from a potential peak of 7 down to 7/3, as shown in Figure 7.10(a). Contrived though this example might appear, I find that the potential for this behavior is exhibited broadly across the SPEC Integer benchmarks: of all the critical instructions that have multiple consumers, more than 50% do not have their most critical consumer first in fetch order.



(a) Reactive load balancing



(b) Proactive load balancing

Figure 7.10. Load-balancing to keep critical dataflow collocated. Each diagram shows a different steering of the same dataflow depicted in Figure 7.9. The target machine is the 8x1w configuration. In the top diagram, a simple dependence-based policy collocates the first (in fetch order) consumer with its producer, and load-balances subsequent consumers. This causes the recurrence dataflow to spread across clusters, incurring a global communication penalty on the critical spine. In the bottom diagram, a *proactive load balancing policy* pushes the first consumer away from its producer, permitting the spine dataflow to collocate.

Figure 7.10(b) shows a more preferable distribution of instructions. The most critical consumer is now retained at the producer's cluster and the less critical consumers are load-balanced. The steady-state ILP is once again 7, and branch resolutions are delayed only two cycles, which is as good as can be done on a machine with 1-wide clusters.

Achieving this schedule is challenging on a machine that performs steering in fetch order. It requires knowing, in advance, which consumer is the critical one and, by impli-

cation, which consumers are not. I will not yet consider specific mechanisms for doing so, since it is not my objective to explore implementation details here. My principal goal at this point is simply to evaluate the potential of proactive load balancing. The results I present in the next section are therefore based on a slightly idealized mechanism, one that would be difficult to build into hardware. That said, a practical scheme does not seem infeasible. An analysis of producer-consumer relationships in my simulator's execution traces reveals two propitious properties of dataflow, one viewed from the producer's perspective and the other from that of the consumer. First, the most critical consumer for a given producer tends to be statically unique: about 80% of all values produced can be associated with a statically unique most-critical consumer. Second, a given consumer tends either to always be the most critical consumer of its producer's value, or is almost never the most critical one. That is, there is a bimodal distribution of static consumers' tendency to be the most critical one. This bodes well for dynamic schemes that train predictors to associate producers with their most-critical consumer, or that tag consumers as the most critical for their operand. Moreover, the stability of these relationships implies static schemes can also do well. This is a subject I explore in detail in Chapter 9.

7.5 Performance results

My objectives in this section are twofold. First, I aim to give empirical evidence in support of my claims that the previously identified causes for performance loss (Sections 7.2 through 7.4) are, in large part, responsible for the IPC difference between state of the art policies and what is potentially achievable (as per Chapter 6). Second, I aim to demonstrate that the policies I have enumerated are indeed effective at mitigating performance problems. To this end, I implemented those policies in my timing simulator and evaluated their performance on the SPEC Integer benchmarks. I am not, in the process, advocating any particular implementation mechanisms. As I said earlier, my goal here is merely to

substantiate my claims quantitatively.

I implemented a likelihood of criticality predictor by tracking the fraction of executions that were detected as critical, using the same critical path detector used in the focused steering and scheduling framework [35]. Other work [82] has demonstrated that stratifying LoC into 16 levels produces results almost equivalent to a counter with unlimited precision. Intuitively this makes sense, since 16 levels generally distinguish two instructions that differ in criticality by as little as 7%; they will fail to select the more critical instruction only when the two have very similar LoC. That same work also demonstrated that using probabilistic counter updates permits implementing a predictor that stratifies 16 levels of LoC using just 4 bits of storage, less space than the 6-bit counters used by Fields *et al.*

With the LoC predictor in place, implementing LoC-based selective stalling is straightforward. As previously noted, I use a fixed LoC value of 30% as a threshold to govern the stalling behavior: any instruction with predicted LoC above that value induces a dispatch stall if its producer's cluster is currently full.

The proactive load-balancing policy is the most challenging to implement in a fully dynamic framework. My implementation works as follows. To decide which consumers should be load-balanced, I track the most critical consumer of each register seen at instruction steering time. When a consumer retires, I compare its LoC with that of the most critical consumer thus recorded; if the value is lower, I tag that consumer as a candidate for proactive load-balancing. On top of this, I modify the dependence-based steering logic to permit just one consumer to follow each producer. This is accomplished by tagging the producer's target register when it has been followed, so that it will be ignored by future consumers. I override this single consumer policy when a particularly critical consumer is encountered: I refuse to load-balance an instruction if its LoC is greater than 5% and it is at least half as critical as the producer (suggesting that it is the most critical consumer). While I do not believe that this constitutes a reasonable implementation, it is one that does not require any oracular knowledge, suggesting that a realistic implementation exists.

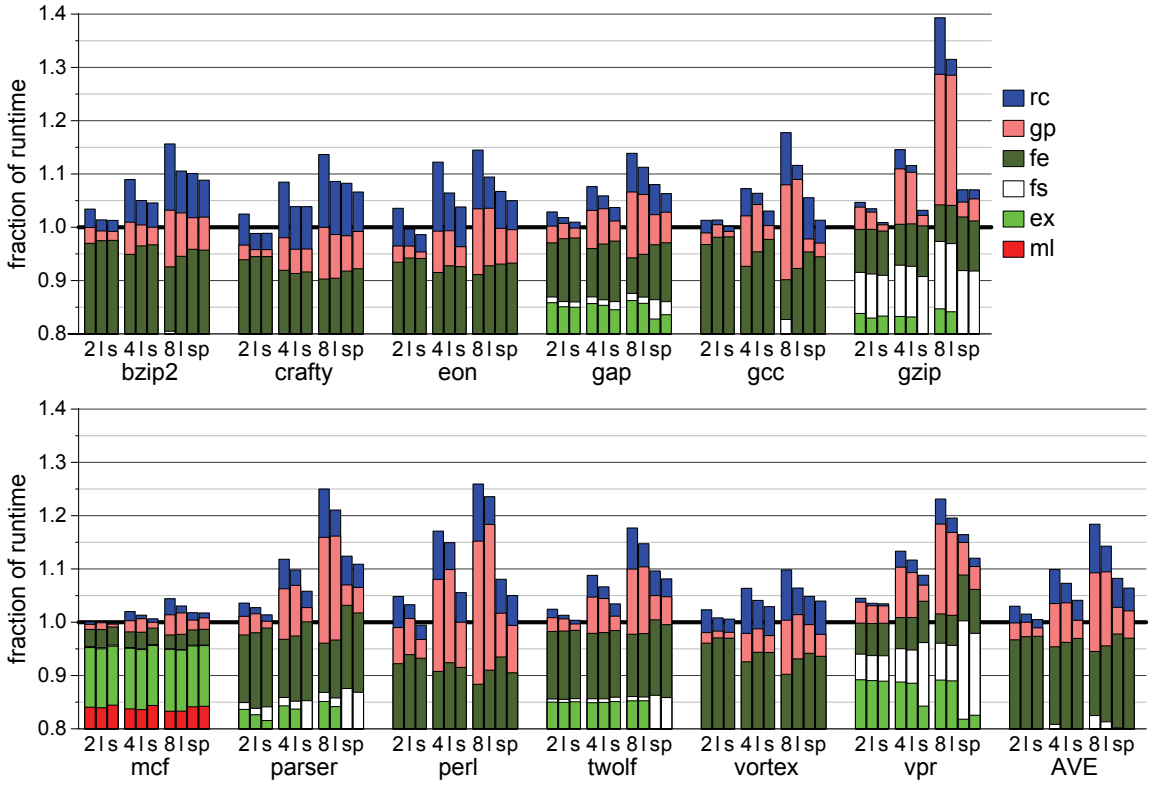


Figure 7.11. Critical path breakdown for LoC-based policies. The first bar for each configuration (labeled '2', '4' and '8') repeats the data from Figure 7.2, which shows the critical path breakdown for the focused steering and scheduling policy developed by Fields *et al.* Bars labeled 'l' substitute LoC for binary criticality, using this more fine-grained measure to prioritize instructions in the issue queue. Bars labeled 's' add LoC-based stalling behavior to the steering logic to prevent load-balancing of high-LoC (execute-critical) instructions. Finally, bars labeled 'p' add proactive load balancing to the 8x1w machine (my implementation of that policy does not benefit the wider clusters). All bars are normalized to a monolithic machine that uses LoC-based scheduling. As before, the global communication penalty is fixed at 2 cycles.

Figure 7.11 demonstrates the benefit of instituting these policies. The LoC-based scheduling scheme is clearly advantageous. For all benchmarks and all configurations it provides a speedup relative to scheduling based on binary criticality. On average, it halves the execution time lost to contention-related stalls and, in some benchmarks (*e.g.* `bzip2`, `crafty` and `vpr`), it indirectly reduces global communication as well.

By contrast, the stall-versus-steer policy is not universally beneficial, but it does provide substantial benefits to `gzip`, `parser`, `perl` and `twolf`, all of which previously saw the biggest penalties from clustering. Much of the massive speedup this policy achieves in `gzip` on the 4x2w and 8x1w machines occurs in long stretches of the execution where

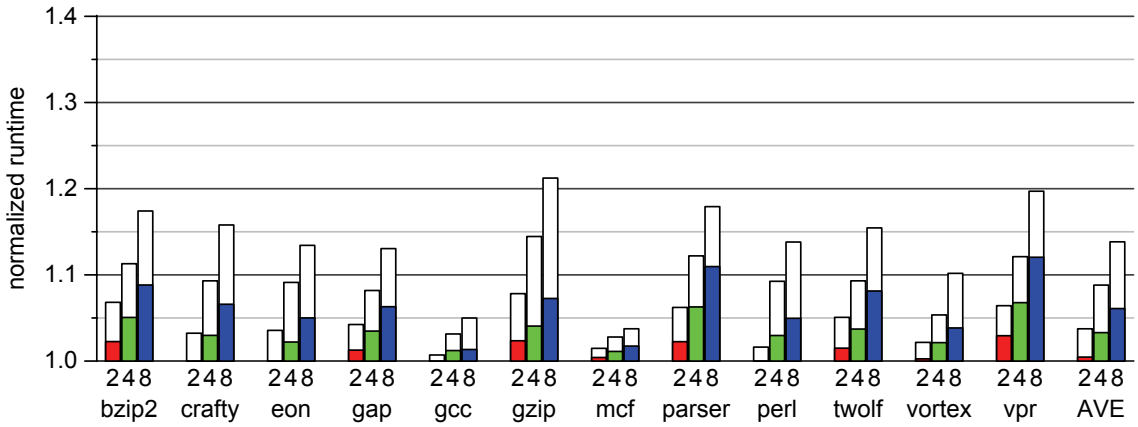


Figure 7.12. Performance of the three LoC-based policies. (*cf.* Figure 7.1, page 125.) The graph summarizes the results from Figure 7.11. Bars labeled 2, 4 and 8 show, for the 2-, 4- and 8-cluster machines, the normalized runtime obtained when all of this chapter's LoC-based policies are combined (proactive load-balancing being applied only in the 8x1w configuration). The shaded portion of each bar corresponds to a 2-cycle global communication penalty; the unshaded portion, to a 4-cycle penalty.

only a fraction of all clusters are used. This confirms my earlier assertion that cluster utilization (*i.e.* workload balance) is not a metric to be optimized. In general, stall-versus-steer leads to a substantial reduction in the contribution of global communication to the critical path signature (the 'gp' category), but it occasionally aggravates the contribution of contention-related stalls (the 'rc' category). This occurs when non-critical instructions that are assigned to the critical cluster are delayed too long, a problem that is mitigated by proactive load-balancing.

As implemented, the proactive load-balancing policy only benefits the 8x1w machine; the other configurations have multiple-issue capabilities at each cluster, so they are less sensitive to load imbalance. This policy improves performance by eliminating contention stalls from near critical paths, reducing both the number of critical contention stalls and the frequency at which near-critical paths are made critical. However, it offers little benefit to programs like `gzip` and `vortex`, most likely because of a lack of easily distinguished non-critical candidates for load-balancing.

Figure 7.12 reproduces the data from Figure 7.11 in a more succinct form. This makes it more clear just how beneficial the new policies are. With the global communication

penalty fixed at 2 cycles, they sustain performance on the 2-, 4- and 8-cluster machines that is within 2%, 4% and 6%, respectively, of the monolithic machine. These correspond to reductions of about 44%, 45% and 60% in the penalties suffered by the focused steering and scheduling policies developed by Fields *et al.* (see Figure 7.1). Moreover, when the global communication penalty is increased to 4 cycles, performance relative to the monolithic machine remains comparatively good, with only the 8x1w configuration now losing more than 10% on average.

These results bring the performance achieved to within 5% of that attained by the idealized list scheduler in Chapter 6. The obvious question now arises as to what prevents removing the remaining 5%. Since I have not attempted to explore the design spaces opened up by each of these new policies, there is almost certainly room for improvement. But this is not likely to account for all of the remaining performance loss. I believe that the bulk of the 5% loss results from an inefficient distribution of ready instructions across the clusters. Although proactive load-balancing pushes subsequent consumers to different clusters, achieving optimal load balance requires that these instructions be assigned to a cluster that does not already have — and will not soon have — ready instructions. In other words, choosing the least-full cluster in these circumstances is not always appropriate.

This requirement is most challenging when the program code exhibits ILP equal to the machine's aggregate issue width. Figure 7.13 shows how effective the policies are at extracting all of the available ILP on the 8x1w configuration. When the available ILP is exactly 8, a clustered machine needs to have distributed one ready instruction to each of its clusters. This is particularly challenging because, in the presence of non-unit latency instructions, it implies the need to assign more than one dataflow chain to a cluster, each with ready instructions on interleaving cycles. Achieving such a fine-tuned load balance would require tracking exactly when and where each instruction will be ready — precisely the problem faced by steering logic in a laned machine. The idealized list scheduler, having an exact view of all in-flight instructions, is able to make these fine-grained decisions. In

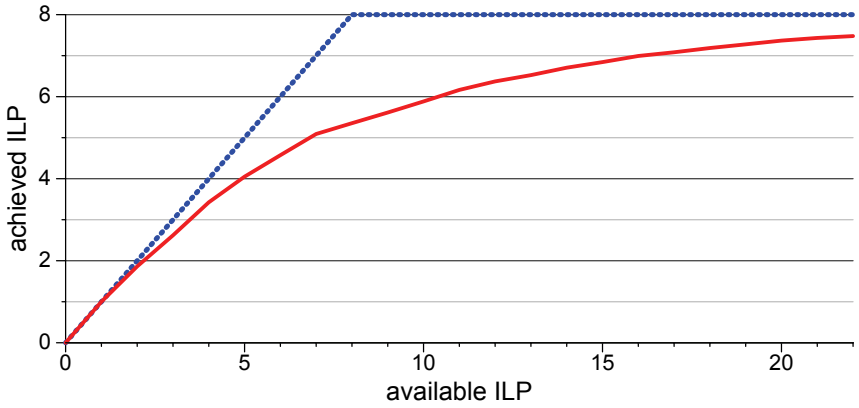


Figure 7.13. IPC as a function of available ILP. The *available ILP* curve is computed on a cycle-by-cycle basis by counting the number of ready instructions across all clusters. The *achieved ILP* curve is the average number of instructions executed in cycles with a given available ILP. Data is averaged over all the SPEC2000 Integer benchmarks for the 8x1w machine.

a realistic machine, steering is decoupled from scheduling, so cluster assignments must be made in the absence of such precise information.

However, the problem becomes much easier when available ILP is either very high or very low. For example, with an available ILP of 24, there are likely to be 3 ready instructions at each cluster, making it unlikely that there will be cycles in which the machine fails to achieve full utilization of its execution resources. Equally, when available ILP is very low, there are few active dataflow chains, so each can be assigned to its own cluster, guaranteeing that ready instructions will be executed every cycle, as per the demands of the dataflow-oriented execution model.

7.6 Conclusion

Motivated by the very good idealized performance potential I demonstrated in Chapter 6, I showed in this chapter that it is also possible to sustain good performance in practice. I started by analyzing the behavior of a state-of-the-art scheme that uses criticality to guide its instruction steering and scheduling decisions. For this purpose, *critical path analysis* has once again proved an indispensable means for identifying — and quantifying — the key sources of performance problems in a given design. In this case, it led to two important

discoveries. First, known-critical instructions suffer contention stalls in spite of being given preferential treatment by the scheduler. Second, load-balancing is responsible for the bulk of global communication observed on the critical path.

Regarding the first of these, I showed that the problem ultimately arises because critical path prediction is not accurate enough to consistently identify which dynamic instruction instances are going to be critical. Accepting that such predictors will necessarily be inaccurate, I then argued that it is the ability to accurately gauge *relative criticality* that is the real objective in any scheme aiming to give preferential treatment to the critical path. An alternate approach uses the *frequency* at which each instruction appears on the critical path, a good indicator of the *likelihood* that its next dynamic instance will in fact be critical. I proposed the *Likelihood of Criticality (LoC)* metric to do just that. Comparing instructions on this basis, always giving precedence to the one with higher LoC, effectively reduces the chances that the wrong dynamic instance will be given precedence. This capability is particularly important in the context of criticality-based scheduling, where pairs of instructions that are both frequently critical, but not at the same rate as one another, are liable to contend for issue slots. Binary criticality fails in such cases simply because it is too coarse-grained to distinguish such instructions. A scheduler equipped with LoC information can, by contrast, hedge its bets by always preferring the one with higher LoC. I showed that such an approach, on average, halves the time lost to contention stalls.

The load-balancing problem poses a more difficult challenge. When programs are in execute-critical regions, and those regions are clearly execute-critical, a stall-over-steer policy can eliminate global communication from long and narrow critical dependence chains. The LoC metric comes in useful here since it can serve as a means for identifying the execute-critical code regions where stalling is appropriate. A stall-versus-steer policy improved the performance of some benchmarks by as much as 20%. I also observed that a stall-versus-steer policy effects a better distribution of ready instructions among the clusters, but found that its potential in this respect is limited by fetch order constraints and by

codes in which a critical dependence chains are not easily discernible. To tackle that problem, I proposed a proactive load-balancing scheme to push non-critical consumers away from their producers, leaving room for the more critical consumers yet to be fetched.

Together, these policies bring the performance of all of the clustered configurations to within about 5% of the idealized scheduling results. Much of the remaining performance gap results from the decoupling of scheduling and steering in a real machine. Specifically, in the absence of a precise and immediate view of instruction readiness, it is difficult for steering to optimally distribute the ready instructions among the clusters. This problem is most pronounced in the 8x1w machine when the code it executes has ILP close to the aggregate issue bandwidth. From an IPC point of view, then, a machine with 1-wide clusters is less appealing than the wider configurations I examined. Since 2-wide clusters are only moderately more complex, I feel they are the more complexity-effective design point.

In terms of their IPC potential, the clustered machines constitute a far more appealing design space than the laned machines. Even a machine with 1-wide clusters — the configuration that performs the worst among those I evaluated here — outperforms by far the heavily idealized laned machines I examined in Chapter 4. Though the laned machines are appealing from an implementation point of view, particularly in terms of their power consumption, that even small out-of-order clusters, whose implementation complexity is bound to be modest, can perform so well constitutes a strong case in favor of clustered designs. Further, with a view to exploring fused-core designs in CMPs, an idea I discussed in Section 4.5 in the context of in-order cores, this chapter's results add to the appeal of CMPs comprising small, out-of-order cores. Prior work has promoted such designs, but has not exploited instruction criticality to manage resource allocation [44].

I view the policies developed in this chapter as a first step in the design of an effective large-window, wide-issue superscalar machine. But a number of challenges still remain. All the policies rely on dynamic critical path detection and prediction logic being added to the processor pipeline. And implementing an LoC-based scheduling policy will no

doubt complicate the dynamic scheduling logic. In fact, even building a circuit that can do dependence-based steering of 8 instructions to any one of 8 clusters each cycle is not easy. Given these hurdles, any scheme that can move some portion of the decision making into software is compelling. This is a subject I discuss at length in Chapter 9. An important part of that work involves moving the entire LoC infrastructure offline, which I describe next.

Chapter 8

Offline critical path analysis[†]

The LoC-based policies I developed in the previous chapter are built on the assumption that accurate criticality data is available at runtime. For that purpose, I relied on hardware similar to the token-based critical path detector and predictor proposed by Fields *et al.* [35]. That logic consumes a non-trivial amount of silicon (14KB, as originally proposed), requires tight integration with the execution core, and involves a significant amount of switching activity and hence power consumption. My focus in this chapter is on circumventing those implementation costs. I will show that they are indeed avoidable: an online critical path infrastructure is, to a large extent, superfluous because instruction criticality can be computed just as effectively offline.

The scheme I have developed replaces critical path hardware with an offline, profile-guided trace fabrication infrastructure. This uses basic profile information, plus a memory image of a program's code segment, to fabricate *random* traces that are representative of that program's execution. Those traces, which are annotated with details on microarchitectural events (*e.g.* branch mispredictions, cache misses), and which include memory alias information, are processed by a simple timing model to identify the static instructions that frequently appear on the critical path. The resulting criticality information proves to be just as effective as that derived from a sophisticated online infrastructure. Figure 8.1 demonstrates this, both for the binary criticality schemes developed by Fields *et al.* and the LoC-based schemes I presented in the previous chapter. Overall, the *static* criticality data derived from my fabricated traces provides 93% of the performance benefits achieved by

[†] The content of this chapter derives from work published at the 2008 International Symposium on Code Generation and Optimization [83]

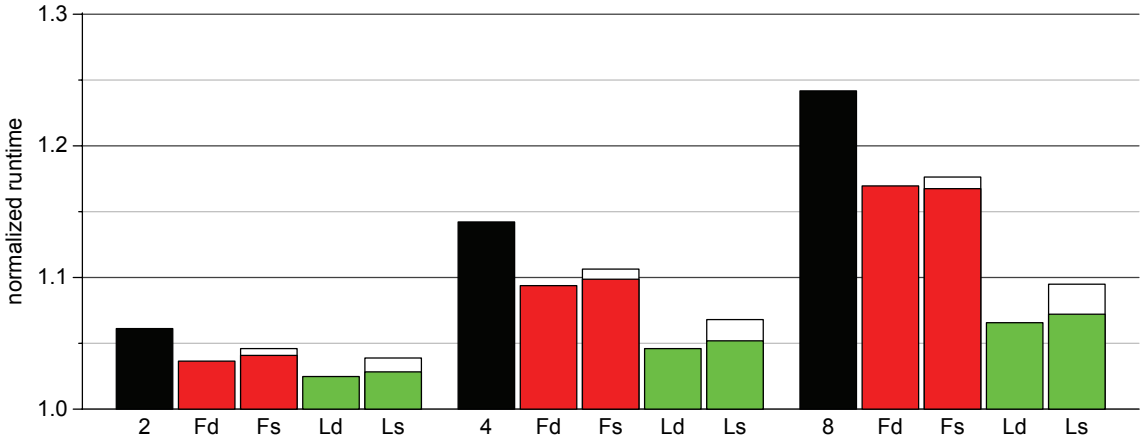


Figure 8.1. The potential of static criticality. The graph shows the runtime of the previous chapter's 2-, 4- and 8-cluster machines relative to a resource-equivalent 8-wide out-of-order superscalar. (More details of the simulation infrastructure, and these performance results, are provided in Section 8.4.) Bars labeled '2', '4' and '8' show, for reference, the performance of a standard dependence-based steering policy, which ignores instruction criticality. The 'Fd' and 'Ld' bars show the impact of taking criticality into account, first using the focused steering and scheduling policies proposed by Fields *et al.* [35] (*cf.* Figure 7.1, page 125), and then using the likelihood of criticality (LoC) policies presented in Chapter 7. Both these schemes rely on critical path detection and prediction logic in hardware. The 'Fs' and 'Ls' bars show *static* versions of those two schemes, where each instruction is tagged with a fixed criticality value derived from the offline trace fabrication scheme. The unshaded portion of those bars shows the performance implications of not fabricating memory dependences in the synthetic traces.

dynamic, hardware-based critical path predictors.

A high-level view of the trace fabrication infrastructure appears in Figure 8.2. Broadly, the scheme comprises two components: *trace fabrication* and *trace analysis*. The former, which I describe in detail in Section 8.2, is the more important of the two, since it is the quality of the fabricated traces that ultimately determines the utility of the criticality information that can be collected from them. Those traces are constructed so as to reflect the same flow of control and the same dataflow relationships typically observed in a real execution, both of which are important for correctly distinguishing the instructions that tend to be critical from those that do not. Control flow is fabricated by means of a profile-guided random walk through the executable. Section 8.2.2 demonstrates the efficacy of this technique.

I use a novel technique for synthesizing memory dependences. While register dependences can be derived directly from a trace once control flow has been fabricated, memory-

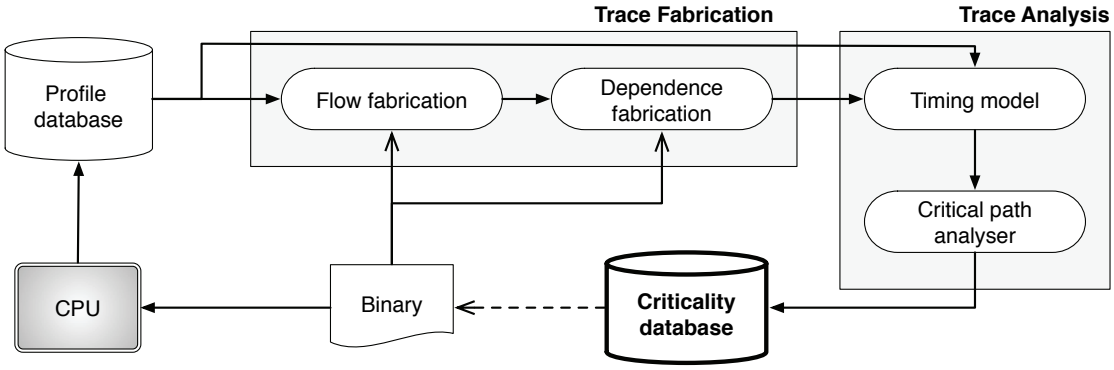


Figure 8.2. The offline critical path infrastructure.

carried dependences — specifically, bypassing store-load pairs — are not obvious from the program itself, nor are they easily profiled. They are, however, important for accurately computing criticality. The unshaded portions of the 'Fs' and 'Ls' bars in Figure 8.1 show that removing synthesized memory dependences from the traces reduces the efficacy of the resulting criticality information to the point that the static schemes now achieve less than 75% of the benefit of their dynamic counterparts. In Section 8.2.3, I show how I use an abstract interpretation technique to compute symbolic values for load and store addresses and, thereby, correctly identify over 95% of all the aliasing store-load pairs.

The trace analysis component, which I describe in Section 8.3, uses a simple timing model to annotate each instruction in the fabricated trace with timestamps reflecting their progress through a simulated pipeline, one that models, at a high level, the same constraints imposed by the host CPU. It also uses profile data to associate microarchitectural events like branch mispredictions and cache misses with specific instructions in the trace, in so doing exposing those instructions, together with their backward dataflow slices, as critical path candidates. The final step — critical path analysis — simply delineates a critical path through the timing-annotated trace, for which purpose I use the dependence graph technique described in Section 1.3.2 (page 29). The resulting *criticality database* records the tendency of each static instruction to reside on the critical path.

Though my focus in this chapter is on mechanisms for generating the criticality database

offline, it is difficult to evaluate the quality of that database in the abstract: its efficacy is determined, ultimately, by the performance that can be derived from it. I therefore use the previous chapter's LoC-based instruction steering and scheduling policies to demonstrate the utility of static criticality data. Section 8.4 presents that study. In addition to validating the trace fabrication approach, the results I present there implicitly demonstrate two properties of criticality that are fundamental to the viability of any such offline scheme. First, because the fabricated criticality is computed without any knowledge of correlated events in the trace (*e.g.* branch B is taken if and only if branch A is taken), one must conclude that such information is largely unnecessary for computing criticality. Second, there is little need for adaptiveness in criticality prediction: the trace fabrication approach computes a single *static* criticality value for each instruction, yet it achieves performance that is very close to that achieved by a hardware predictor. In this regard, I further find that an instruction's criticality is largely independent of program input, permitting previous program inputs to be used to predict instruction criticality.

Though my focus in this chapter is specifically on generating LoC data without hardware support, the trace fabrication scheme could be useful in a much broader context. First, being an effective mechanism for delineating the critical path, any scheme that benefits from criticality is liable to also benefit from trace fabrication. Figure 8.1 showed this to be the case for the binary criticality scheme from Fields *et al.*, but any criticality-driven schemes for optimizing complexity-effective implementations of aggressive microarchitectures would likewise benefit [34, 67, 98, 99]. There is also potential to use trace fabrication as a postmortem tool for analyzing program and microarchitectural bottlenecks. Though I will not explore such avenues here, I believe it would be easy to extend the infrastructure so that it can be used to compute a critical path signature — or, equivalently, a *CPI stack* [29] — for a profiled program — in the manner of the bottleneck analyses I presented in Chapters 3 and 7. I will return to this subject at the close of the chapter. In general, trace fabrication represents a "sweet spot" between the extremes of detailed microarchi-

tectural simulation and abstract program analysis. With respect to the former, it can be performed more efficiently and requires neither building a complete simulator nor having to ensure that a representative sample of the program is simulated. With respect to the latter, it naturally incorporates both microarchitectural and dynamic dataflow information into the analysis, which program analysis does not (easily) capture.

8.1 Background

An online critical path detector and predictor is appealing in two respects. First, it is flexible enough to adapt to changes in program behavior, and can therefore improve the efficacy of whatever pipeline optimizations use its criticality predictions. Second, since those optimizations will frequently change the critical path itself (*e.g.* value prediction can shorten it), an online scheme provides the ability to respond to the very optimizations it drives. Unfortunately, those benefits come at a cost. The token-based predictor from Fields *et al.*, for example, requires a multi-ported 1.5KB array for detecting critical instructions, plus modifications to the execution core to record very detailed information on the relative timing of various events in an instruction's lifetime. And a critical path predictor, which is trained by the token-passing detector logic, itself requires a table of 16K 6-bit saturating counters for binary criticality prediction; an LoC predictor would require even more than this, though recent work has shown that it can be optimized, without significant loss of precision, to use 3 bits for each predictor table entry [82]. Regardless, an online critical path infrastructure incurs a significant cost — enough, perhaps, to mitigate the benefits of the criticality-aware optimizations it enables. My principal objective in this chapter is to demonstrate that sophisticated online infrastructures are superfluous. Accurate criticality information can be derived offline using a profile-guided synthetic trace fabrication scheme, the details of which I describe in Sections 8.2 and 8.3. First, however, I briefly review related work.

Given that profiled program behavior plays a central role in my trace fabrication infrastructure, my approach shares a number of similarities with previous work that uses profile data to reproduce statistically similar behavior in an offline context. For example, statistical modeling [45, 70] and statistical simulation [71, 74] both make use of probabilistic models to reproduce, in aggregate, the kind of behavior observed by the real program. While I likewise exploit probabilistic properties of program behavior, I differ fundamentally from those prior studies in that an *aggregate* result is not sufficient. Specifically, I am not interested in reproducing an IPC figure offline, but rather in examining dynamic sequences of instructions that are representative of the real execution. Moreover, I need to do so in such a way as to be able to correctly attribute criticality to the specific static instructions that are responsible.

My work is most closely related to the *shotgun profiling* technique proposed by Fields *et al.* [34]. That scheme also uses profiled behavior to feed an offline analysis of instruction criticality. However, whereas I profile only very basic dynamic events (details in the next section), shotgun profiling snapshots very low-level events pertaining to an instruction's progress through the pipeline. Though simpler than a full online critical path detection infrastructure, shotgun profiling still constitutes a substantial addition to the pipeline. I obviate the need for that hardware by relying much more on the original program binary to ensure that fabricated traces embed the same dataflow patterns seen in a real execution.

8.2 Trace fabrication

In this section, I describe my trace fabrication infrastructure in detail. I noted earlier that it is important that the fabricated instruction traces be representative of those observed in a real execution. I mean by this that the traces should embed the same dataflow patterns that are typically present in a real execution and, more importantly, that they facilitate mapping the important dynamic dataflow patterns back onto the static instructions that induce them.

This requirement is conveniently viewed as comprising two parts. The first, and the more important of the two, is *control flow fabrication*. Section 8.2.2 describes how I use a profile-guided "random walk" through the program binary to fabricate the synthetic flow of control. The second component is *data dependence fabrication*, which I describe in Section 8.2.3. My focus there is on fabricating dependences through memory, since dependences through registers are naturally induced by the fabricated control flow. Since profile data seeds this whole process, I begin in Section 8.2.1 with a brief discussion of the profiling infrastructure I rely on.

8.2.1 Profiling infrastructure

The trace fabrication infrastructure is driven by a database of profiled program behavior (recall Figure 8.2). Since that database contains information on microarchitectural events like cache misses and branch mispredictions, I rely on hardware support for profiling. My requirements in this regard are modest, however. I need to precisely attribute events to the instructions that cause them, but this is a feature already available in many commercial microprocessors [24, 63, 69], as well as proposed for *AMD*'s recently announced *Lightweight Profiling* [2]. I will therefore not describe here exactly how the profile information is collected; more important is the specific information I rely on.

Figure 8.3 shows the program properties that are profiled. The bulk of the profile database is just a histogram of sampled PCs. In addition to that, it records counts of cache misses and branch mispredictions for each static load and branch that is sampled. These are used by the trace analysis part of the infrastructure (Section 8.3). I also maintain a histogram of branch targets for each branch instruction that is sampled. This is used by the control flow fabrication logic described in the next sub-section.

I impose two requirements on the profile data. First, like any profile-based technique, I require that a profile cover the program's code footprint with enough samples to permit ready identification of the hot portions of the code. I characterize the sensitivity of my

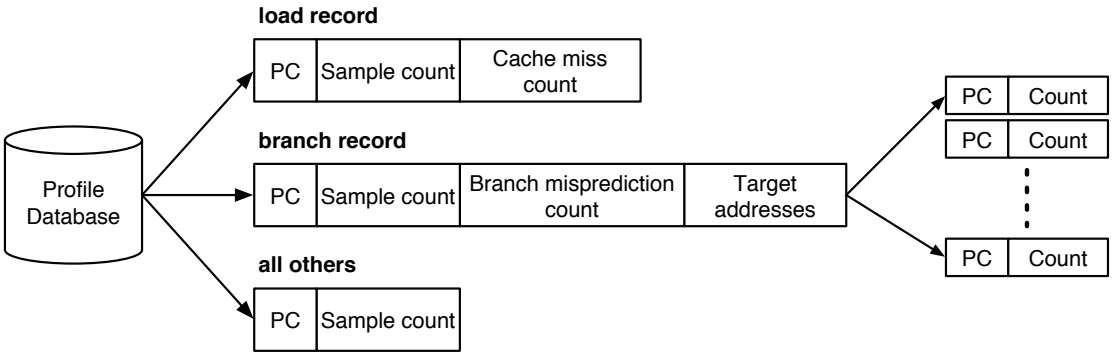


Figure 8.3. Profile data. There are 3 types of records in the profile database: one for load instructions, one for branches, and one for all others. All record a sample count for the corresponding static PC. Loads and branches also maintain a record of microarchitectural events (cache miss or branch misprediction) for that static instruction. A histogram of branch targets for each branch is also maintained.

results to the quantity of profile information in Section 8.4. In principle, the requisite sample count can be achieved either by profiling over a short period of time with a high sampling rate or over a longer period of time with a low sampling rate. Second, I require that sampling be unbiased, in the sense that the number of samples taken for a given static PC be proportional to the relative frequency at which that instruction executes. This ensures that the trace fabrication logic will produce traces whose code coverage corresponds to that in a real execution.

8.2.2 Control flow fabrication

The fidelity of my fabricated criticality information is determined, first and foremost, by the extent to which the traces faithfully reproduce the control flow observed in the real program. This is because control flow induces dataflow, and dataflow has a first-order effect on the critical path. Put another way, if one gets the control flow right, one immediately gets most of the dataflow right, as register dataflow is derived directly from control flow.

Algorithm 1 Control flow fabrication

```
 1: callStack ← ∅
 2: currPC ← GenerateRandomSeed()
 3:
 4: for i ← 0 to TRACELET_LENGTH do
 5:     currInst ← ReadFromBinary( currPC )
 6:     nextPC ← currPC + 4
 7:
 8:     if currInst is a call instruction then
 9:         callStack.push( nextPC )
10:         nextPC ← GuessTarget( currPC )
11:     else if currInst is a return instruction then
12:         nextPC ← ( callStack.empty() ? GuessTarget(currPC) : callStack.pop() )
13:     else if currInst is a branch or jump instruction then
14:         nextPC ← GuessTarget( currPC )
15:     end if
16:
17:     AddToTrace( currInst )
18:     currPC ← nextPC
19: end for
```

Flow fabrication logic

Algorithm 1 shows the main steps in the flow fabrication logic. I operate at a granularity of *tracelets* — short sequences of instructions seeded by randomly picking a starting PC from the profile database. The trace analysis component of the infrastructure, which I describe in Section 8.3, repeatedly consumes these tracelets in order to delineate the critical path.

Random seeding. The `GenerateRandomSeed` function used by Algorithm 1 (line 2) is responsible for picking the PC that will seed the tracelet generation. It operates by randomly selecting a record from the profile database, weighting the choice of an instruction by its sample count. In this way, I tend to seed traces in hot code regions. The ensuing control flow fabrication, described next, ensures that tracelets tend also to remain in hot regions.

Random walking. The for-loop in Algorithm 1 (lines 4–19) constitutes a profile-guided random walk through the program binary. I use the program binary to guide sequential control flow (lines 5 and 6) and the profile database to randomly pick the successors of control flow instructions (lines 8–15). For function `return` instructions (line 12), I maintain a simple call stack from which I pop target addresses; `call` instructions push their successor onto this stack (line 9). In this way, I establish semantically meaningful control flow across function call boundaries. The target addresses for `call` instructions (line 10), direct and indirect `jump` (unconditional) instructions, and conditional `branch` instructions (line 14), are all computed by the `GuessTarget` function.[1] Analogous to random seed generation, `GuessTarget` randomly picks a target PC from among all the profiled targets for the given control flow instruction, weighting the choice by those targets' sample counts.

Discussion

The above logic ensures that the tracelets are very likely to mimic real program control flow, repeatedly iterating through hot loops and tending to follow the most biased conditional paths within them. To evaluate the extent to which this is indeed happening, I draw on previous work in the area of *path profiling*. I use a path profile as a succinct measure of the control flow embedded in a given trace, and hence for comparing the fabricated traces to those from a real execution: the more closely the two match, the more accurately the fabricated traces reflect real execution. I follow the approach taken by Ball and Larus, who define a path as an acyclic sequence of intra-procedural basic blocks [7]. Unlike them, however, I do not use the so-called Wall weight-matching metric to compare two path profiles, preferring instead to use the *overlap metric* employed by Arnold and Ryder [5]. This is a more intuitive means for comparing two path profiles, and it is readily extended to also quantify the accuracy of dataflow dependence fabrication, which is discussed in

[1] Unconditional, direct (PC-relative) jumps are a special case. These have a statically-fixed target instruction that can be discovered from the binary.

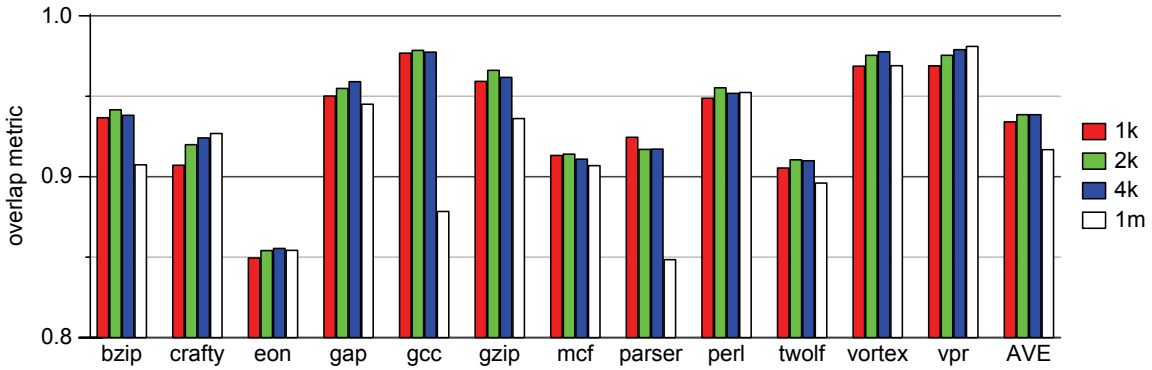


Figure 8.4. Path profile overlap. Moving left to right in each group, bars '1k', '2k' and '4k' plot the value of $M_f(P_r, P_f)$ for tracelet lengths of 1-, 2- and 4-thousand instructions; the '1m' bars correspond to 1-million instructions per tracelet. Each plotted value is computed by running trace fabrication for a total of 60-million instructions (60-thousand tracelets for bars labeled '1k', for example). In all cases, the profile databases driving the trace fabrication logic are collected from the same real program trace against which the fabricated traces are compared.

Section 8.2.3.

Very briefly, the overlap metric is defined as follows. The flow along path p in path profile P, denoted $f(p, P)$, is the the total number of times path p occurs in profile P. The coverage of p in P, which is the fraction of total flow in P that can be attributed to p, is then given by $c(p, P) = f(p, P)/f(P)$, where $f(P) = \sum_{p \in P} f(p, P)$. The flow overlap metric, denoted M_f, compares real-trace path profile P_r to fabricated-trace profile P_f as follows: $M_f(P_r, P_f) = \sum_{p \in P_r} min\{c(p, P_r), c(p, P_f)\}$. This quantifies the extent to which the fabricated profile agrees with the real one on the fraction of total flow that is attributable to each of the real profile's constituent paths. Two identical path profiles will yield an overlap value of exactly 1. Any differences between the two will yield a value less than that: too little (or no) coverage of a given path by the fabricated trace will be captured by the min operation; too much coverage will be capped by the min, and will ultimately be penalized by the ensuing deficit of flow along other paths.

Figure 8.4 plots the value of the overlap metric for the SPEC Integer benchmarks. That data is collected by first generating a real instruction trace (one derived from a program execution) of a given length. From this, I compute P_r, the real-trace path profile. In addition, I generate a profile database for the real trace, and use that to fabricate a collection

of tracelets whose aggregate length equals the size of the real trace. I then compute P_f for those fabricated traces. The data in Figure 8.4 shows the value of $M_f(P_r, P_f)$. Since the fabricated trace is being compared to the same one from which its profile database is derived, these results constitute a form of self-training. I do so at this point to avoid clouding the results with artifacts of profiling specifics, but I will show later (in Section 8.4) that removing these idealized assumptions has little impact. The graph confirms that the fabricated traces are remarkably effective at reproducing the control flow observed in a real execution. On average, close to 95% of the real trace's flow is matched by the fabricated traces. This data is consistent with results published by Ball *et al.* [7], who found that a greedy strategy for deriving a path profile from an edge profile (*i.e.* a profile of branch targets, like the one I rely on), is able to very closely match the real path profile. This is testament to the highly-biased nature of most control flow.

The graph also shows that longer tracelets marginally improve the accuracy of the path profile, but only to a point. The '1m' bars, for example, show that very long tracelets generally yield a worse overlap value, and in some cases show marked degradations (`gcc` and `parser`). This is because, as tracelets grow in size, there is a reduction in the total number of times the fabrication logic is reseeded. Using fewer seeds increases the chance that the random walk ventures down a cold path and remains there too long. This can distort the extent to which the traces correctly reflect the relative time spent in each code region, as well as the likelihood that they properly cover all the executed regions. My approach of repeatedly reseeding the flow fabrication is key to avoiding such problems. In this regard, very short tracelets are most appealing, but this requirement must be balanced against the need for sufficiently long traces to capture all the dataflow and microarchitectural constraints that have bearing on the critical path. Tracelets shorter than the machine's ROB size, for example, will fail to capture the C–D edges in the dependence graph model. From empirical evaluation of various alternatives for tracelet length, I find that 1,000 instructions strikes a good balance between the above factors, as well as minimizes the number of in-

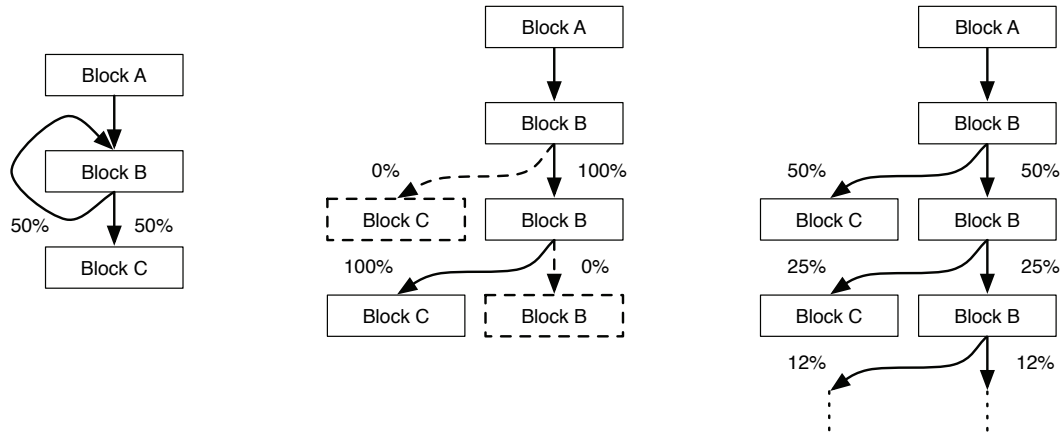


Figure 8.5. Correlated flow in `eon`. (a) The static control flow graph is distilled from hot method `ggSpectrum::Set`. Edges are annotated with their profiled frequencies. (b) Only one path through this code is ever observed at run time: exactly 2 iterations through the loop. (c) The fabrication logic tends to explore all possible paths through the code.

structions that must be analyzed before sufficiently accurate criticality profiles are obtained. I will therefore confine my analysis in the remainder of this chapter to tracelets of 1,000 instructions each.

Figure 8.4 also shows that the flow fabrication is not uniformly successful across all benchmarks. The `eon` program, in particular, stands out. In that case, the less effective path coverage arises because of *correlated* control flow, which the fabrication logic does not take into account. Figure 8.5 shows why this is a problem in `eon`. Only a subset of all possible paths is observed during execution of this code, but the trace fabrication approach tends to enumerate all of them, thus watering down the path profile. These effects are comparatively rare, however, and, as I will show, they tend to have little impact on the integrity of the criticality that is extracted from the fabricated traces.

8.2.3 Data dependence fabrication

From a critical path point of view, dataflow dependences through the register file are by far the most important. Because I fabricate traces based on actual instructions read from the

binary (Algorithm 1, line 5), I faithfully model all intra-block register file dataflow. And because most of the control flow is representative of real execution, inter-block dataflow is mostly accurate too.

A more challenging problem is posed by dataflow through memory. As I showed in Figure 8.1, capturing the main store-to-load communication exhibited by the program is important for accurate identification of critical instructions. I therefore require a mechanism for fabricating aliasing behavior in the traces. To be clear, I make a distinction between *memory aliasing* and *memory bypassing*. A store and load alias if they touch the same address and there is no intervening store to that address; a store bypasses to a load if the pair alias *and* the store is still in the pipeline when the load issues. That is, bypassing is a pipeline-specific notion; aliasing is a property of the program. The goal at this point is to generate synthetic *alias* information, not bypass information. The trace analysis logic, which I discuss in Section 8.3, uses that aliasing information, together with knowledge of pipeline parameters, to decide whether a given store should bypass to a given load.

In the absence of actual input data, one cannot, in general, know the exact memory addresses generated by a program. However, what matters most in determining memory aliasing behavior is not the addresses themselves, but simply whether two addresses are the same. The approach I take, therefore, is to perform an abstract execution of the tracelet using arbitrary values. Logically, I maintain a simple record of the machine's architected state, initializing every register and memory location with a random value, and then emulate the execution of each non-control flow instruction in the tracelet.[2] I record the "symbolic" addresses thus generated for each memory operation and later compare those values to identify aliasing store-load pairs.

Proceeding in this way, computation within the trace is internally consistent; it is merely seeded with random input. Loads and stores that tend to alias in a real execution are therefore liable to also alias in the symbolic execution, albeit via meaningless addresses. Callee

[2]In practice, I only generate random values on demand, when a storage location that has not yet been written by the tracelet is read for the first time.

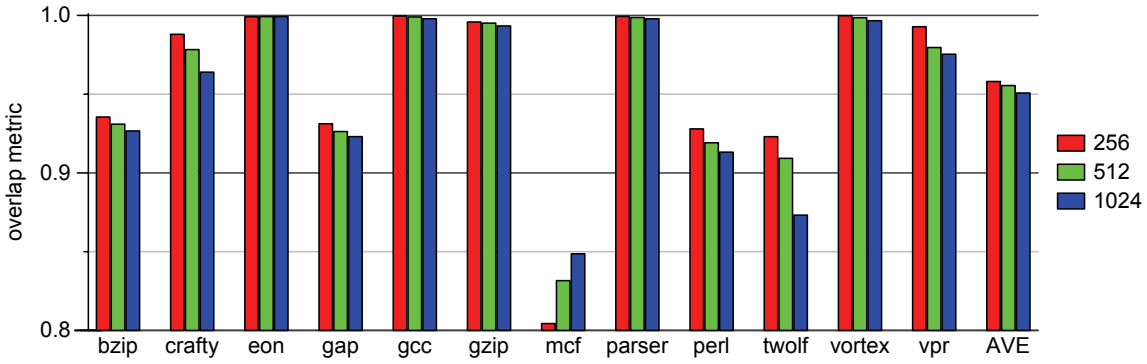


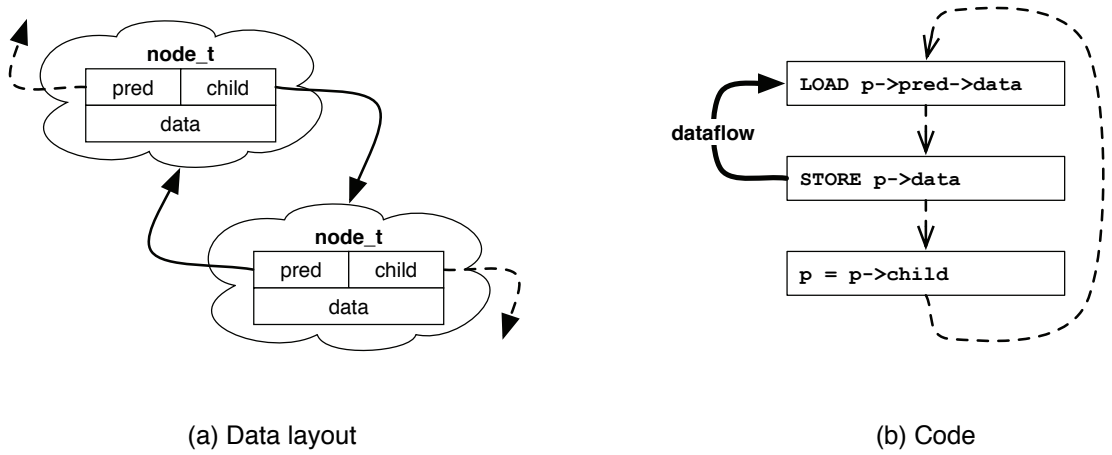
Figure 8.6. Load alias profile overlap. Each bar plots the value of the alias overlap metric for a real program trace relative to one whose memory addresses have been replaced with those generated by the symbolic execution technique. That is, the real trace is decorated with symbolic addresses; the resulting alias profile is then compared with that of the original trace. Aliasing is measured in windows of 256, 512 and 1024 contiguous instructions, which slides across traces comprising a total of 60-million instructions.

saved and restored registers, in particular, are easily detected: I seed the stack pointer with a random value, and subsequent saves and restores, which use fixed offsets relative to that random value, will correctly alias. A number of other aliasing patterns are likewise caught.

To get a feel for the efficacy of this technique, I develop the notion of an *alias profile* for an instruction trace. This is simply a histogram that associates a count with each static load-store pair that participated in a dynamic aliasing. If (st, ld) is one such pair, I define $c(st, ld, A)$ to be the fraction of all dynamic aliasing instances in alias profile A that are accounted for by that pair. This leads naturally to an aliasing overlap metric, the analog of the path profile metric M_f. Specifically, for real and fabricated alias profiles A_r and A_f, I define the alias overlap $M_a(A_r, A_f) = \sum_{(st,ld)\in A_r} min\{c(st, ld, A_r), c(st, ld, A_f)\}$.

Figure 8.6 plots the value of this metric for the alias profiles generated by the symbolic execution technique.[3] Overall, the data shows that the symbolic execution technique is remarkably effective at reproducing almost all the aliasing behavior observed in a real execution, with the overlap metric averaging about 95% across all the benchmarks. Moreover,

[3] Actually, I compute a bidirectional overlap, since the metric, as defined, will falsely report good overlap if, for example, an alias profile deems all load-store pairs to always have aliased. To obviate such problems, Figure 8.6 plots the average of $M_a(A_r, A_f)$ and $M_a(A_f, A_r)$. The path profile overlap is not prone to this problem because total flow in the two profiles is fixed, so too much flow along one path will always be compensated for by less flow elsewhere; not so with alias profiles.



(a) Data layout (b) Code

Figure 8.7. Correlated addresses in `mcf`. (a) The doubly-linked `node_t` data structure induces mutually dependent values in the `pred` and `child` fields. (b) Simplified version of an inner loop in hot method `refresh_potential`, which iterates over the linked structure, updating each node's fields based on the results of computation on its predecessor.

the efficacy of the technique extends to large regions of contiguous instructions, rendering it useful for studying the effects of aliasing even in large-window machines.

Figure 8.6 exposes `mcf` as an outlier — the only benchmark to consistently score below 90%. Figure 8.7 shows why this is occurring. The problem here is analogous to the correlated control flow problem: certain values in a program's data structures are correlated, in the sense that program semantics establish some form of relationship between them. This becomes important for aliasing when those data values are pointers. In the `mcf` code, such a relationship exists between the `pred` and `child` fields of the nodes in a doubly-linked data structure.[4] With the symbolic execution technique, I seed values for these two pointer fields randomly, so I fail to establish any relationship between them. As Figure 8.7 shows, one of `mcf`'s hot loops performs an update to this linked structure, doing so in such a way that updates to the fields of one node are used during the computation on the child node in the subsequent loop iteration. There is, as a result, frequent store-to-load bypassing across successive loop iterations. Fortunately, scenarios such as this are comparatively rare in the

[4]Specifically, the two pointer fields are mutually referential: if `n` points to a non-leaf node, then `n->child->pred == n`.

programs I study, so they do not perturb the criticality analysis very much.

8.3 Trace analysis

In this short section, I briefly describe trace analysis, the second component of the offline infrastructure. Recalling Figure 8.2, this comprises two main components: the timing model and the critical path analyses.

8.3.1 Timing model

The timing model consumes one tracelet at a time. Its task is to annotate each instruction with timing information to mimic the effects of that instruction's progress through a simulated pipeline. Those timestamps reflect the manner in which the fabricated dataflow is likely to interact with the low-level microarchitectural constraints imposed by the host CPU. It is that interaction which ultimately determines the critical path.

The principal requirement I impose on the timing model is that it faithfully model the microarchitecture of the host CPU. This does not mean I need a very detailed microarchitectural simulator, but rather that I need to capture the first-order effects that have the most influence on the critical path. Specifically, I need to model the machine constraints captured by the dependence graph model depicted in Figure 1.2 (page 30): dispatch into the out-of-order window, execution, and exit from the window. This leads to the high-level model depicted in Figure 8.8. The timing model is essentially a simple trace-driven simulator for an out-of-order machine. The main microarchitectural constraints it captures relate to peak instruction supply (fetch) rate, issue bandwidth (different rates for different instruction types), and peak commit bandwidth. It also models the effects of finite buffering capacity in the machine's out-of-order window. Instruction execution latencies are also modeled, but not necessarily true to the latencies imposed on the host CPU. Rather, the objective is to capture the effects of long-latency floating point operations and variable-latency load

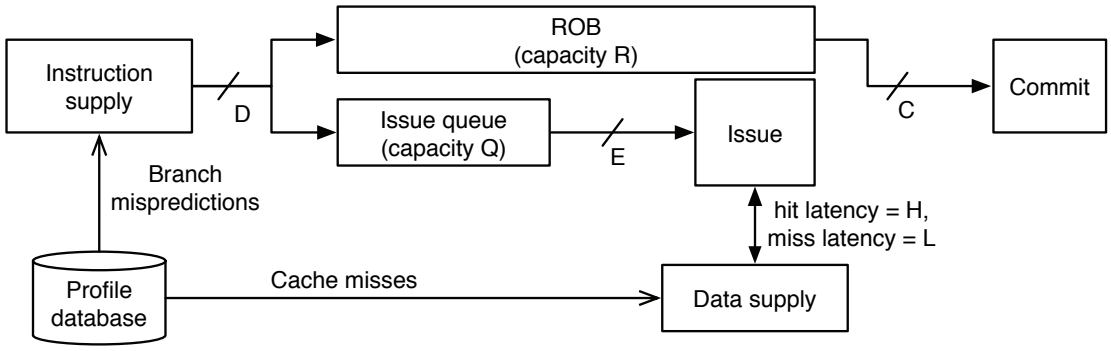


Figure 8.8. High-level view of the timing model.

instructions, and to distinguish those from single-cycle integer operations.

Also shown in Figure 8.8 is the use of profile data to inject branch mispredictions and cache misses into the trace. These microarchitectural events have a strong influence over which static instructions tend to appear on the critical path. The timing model captures their effects by randomly selecting the dynamic instances of branch (load) instructions that will incur a misprediction (cache miss). This is done as per the profiled behavior of the corresponding static instructions. That is, branches that are frequently mispredicted in the real execution will tend to incur many simulated mispredictions in the timing model; likewise for cache-missing loads. I simulate a branch misprediction simply by stalling instruction supply for a fixed number of cycles. Simulated cache misses result in load instructions incurring a fixed number of cycles of additional latency.

My approach of randomly deciding when branch mispredictions and cache misses occur will, of course, miss the effects of correlation among such events, as well as any tendency they might have to cluster in time. However, these effects are of secondary importance. What counts most is that branches that tend to mispredict, and loads that tend to miss in the cache, are frequently flagged as such in the simulated trace, and hence that they and their backward dataflow slices are liable to be exposed on the critical path.

In the empirical work I present in Section 8.4, where I evaluate the criticality infrastructure in the context of clustered superscalar machines, the CPU I use in the timing model

is slightly different from the one depicted in Figure 8.8. In particular, its issue queue is partitioned, and dataflow between those partitions incurs a global communication penalty. I likewise need to model an instruction steering policy for distributing instructions among the clusters. While these components add a little more complexity to the timing model, they do not fundamentally change the basic infrastructure described above.

8.3.2 Critical path analysis

As I noted above, I use the dependence graph model from Fields *et al.* to delineate the critical path. I conduct the analysis on a per-tracelet basis, delineating the entire critical path for each before moving onto the next. Since I have each of the tracelets available in its entirety, I do not need to actually construct the whole dependence graph. Instead, I simply move backwards through each tracelet, using the embedded timestamps to (logically) follow the last-arriving edge at each of the implied nodes. Any instruction whose execution (*i.e.* its E-node) is reached by means of this backward traversal is deemed critical. For each static instruction thus identified, I increment two counters, one for the total number of dynamic instances seen, and another for the total number of critical instances seen; for all other instructions, I increment just the former counter.

Maintaining two counters for each static instruction in this manner permits me to derive both a binary criticality and a likelihood of criticality (LoC) profile from the analysis. In both cases, I use the ratio of critical to total instances to determine the final criticality value of each instruction. To obtain a binary value, I use a threshold ratio of 0.125 to make the not-critical/critical distinction. This corresponds roughly to the mechanism employed by the counter-based predictor proposed by Fields *et al.*, which has its counters incremented by 8 when training critical, and decremented by 1 when training not-critical. Though different threshold values may yield slightly better results, I find that the 0.125 serves adequately for my purposes. To derive an LoC value from the criticality profile, I simply take the ratio of critical to total instances.

Once multiple tracelets have been processed in this way, I write the entire criticality database to file. Obviously, the number of tracelets that are processed before this occurs will have an effect on the accuracy of the resulting criticality database. In particular, processing more tracelets improves the extent to which the complete binary is covered. I present data in the next section to quantify this effect, and hence to gauge the amount of offline tracelet processing that is necessary before sufficiently accurate criticality databases are obtained.

8.4 Evaluation

The previous sections have demonstrated that the fabricated traces closely match real program traces, both in terms of the control flow they embed and the dataflow they induce. Of course, the ultimate utility of the scheme is determined not by the integrity of the traces themselves, but by the quality of the criticality that can be derived from them. As a vehicle for evaluating this, I use the LoC-based instruction steering and scheduling policies that were developed in Chapter 7. I show in Section 8.4.2 that the criticality generated from self-trained profile databases can be used to very good effect in static versions of those schemes. In Section 8.4.3, I then proceed to remove some of the idealized assumptions about the profile databases. I show that the quality of the criticality generated is not very sensitive to changes to the input of the program being sampled, nor to the rate at which samples are collected. Finally, in Section 8.4.4, I quantify the amount of work that must be performed by the fabrication infrastructure before the criticality it generates is of sufficiently good quality. Before I present those experiments, I briefly describe the empirical framework I use in this section, which differs slightly from what I have been using up to this point.

8.4.1 Empirical framework

As before, I use my trace-driven timing simulator to evaluate three clustered microarchitectures, each of which is a different partitioning of a monolithic 8-wide out-of-order superscalar. The monolithic machine, which I again use as the baseline against which to compare the clustered machines, is configured exactly as per Table 5.1 (page 110).

I once again simulate the SPEC2000 CINT2000 benchmark suite. However, whereas I used reduced input sets in previous chapters, all results presented here are obtained from the reference input sets. In Section 8.4.3, I will also use the training input sets to generate the profile databases. Also in contrast to previous chapters, I now perform the simulations at 10 equally-spaced checkpoints across each benchmark's complete run. In each of those runs, I simulate 100-million instructions after warming up the memory system and branch predictor, giving a total of 1-billion instructions simulated per benchmark. When collecting the profiles, I produce one database for each benchmark, this being the aggregation of the behavior profiled during the 10 checkpoint runs.

I have used the trace fabrication infrastructure to drive both my own LoC-based policies (from Chapter 7) and the focused steering and scheduling policies from Fields *et al.* Incorporating the fabricated criticality into those schemes simply involves tagging the corresponding static instructions with a *fixed* criticality value, either binary or LoC, as appropriate. Instructions for which the fabricated criticality database has no data — because the trace fabrication failed to achieve the requisite code coverage, for example — are treated as having zero criticality. For the sake of expediency, I will present data in this section only for the LoC-based policies. Nevertheless, the observations I make apply equally to the binary criticality schemes. Moreover, average results for the binary schemes have already been presented in Figure 8.1.

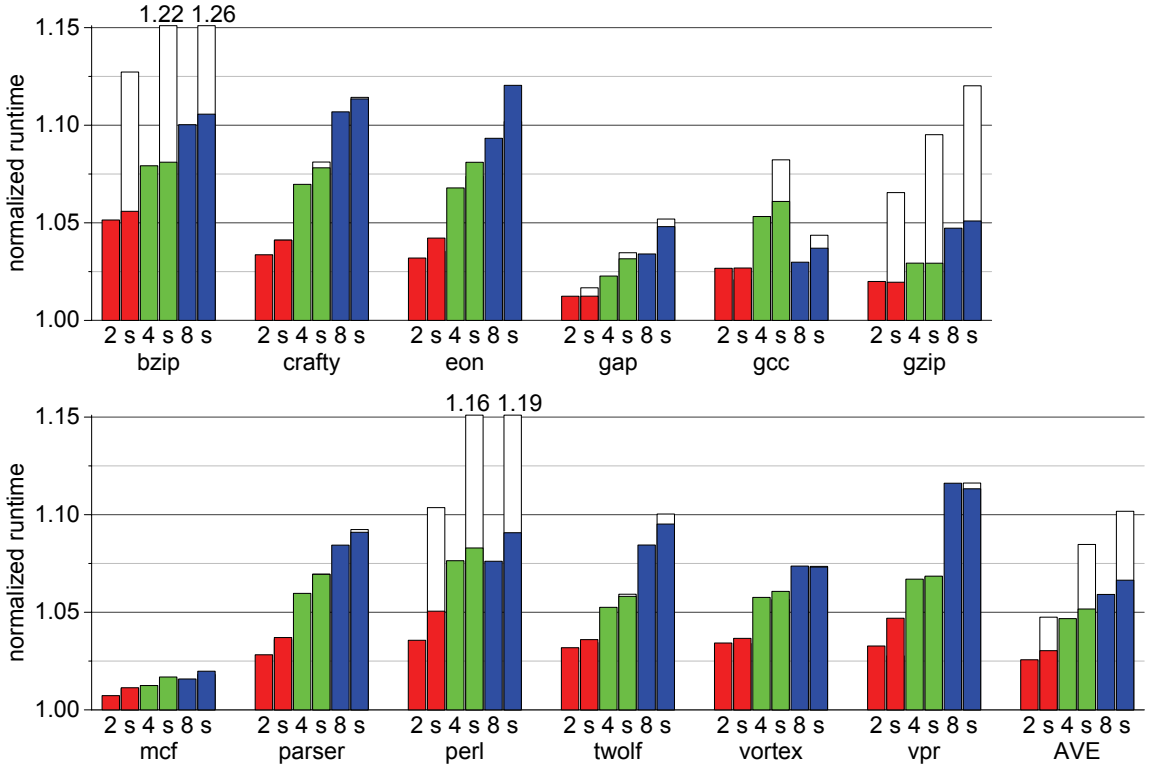


Figure 8.9. The potential of static criticality. The graph plots the runtime, relative to the monolithic baseline, of the LoC-based policies on three different clustered machines. The SPEC reference input sets were used in all simulations. Bars labeled '2', '4' and '8' plot the performance of *dynamic* versions of the LoC-based policies for 2-, 4- and 8-cluster machines. Adjacent to each of those, bars labeled 's' plot the performance of the corresponding machine, driven now by *static* LoC data derived from the trace fabrication scheme. The shaded portions of those bars show the performance achieved when trace fabrication is driven by profiling of the reference inputs (*i.e.* self-training); the unshaded portion (not visible in most bars), by profiling of the training input sets.

8.4.2 Performance from self-training

So as to isolate the effects of extraneous variables, I begin with an evaluation of criticality derived from *idealized profiles*. By this I mean two things. First, the profile database used by the offline scheme is generated by sampling every instruction in the real execution. I thereby ensure complete code coverage. Second, I evaluate the quality of the resulting criticality by using it in simulation of the same traces that were profiled — I self train, in other words. Section 8.4.3 explores the implications of removing these idealized assumptions.

Figure 8.9 shows how the static version of the LoC scheme compares to its dynamic counterpart. Results for the binary criticality schemes of Fields *et al.* (not shown) follow

exactly the same trends. In both cases, the average performance differences between the static and dynamic schemes are extremely small. In fact, the static versions are never more than 3% slower (`eon` on the 8-cluster configuration) and, on average, are less than 1% slower than the dynamic version. That `eon` stands out a little can be accounted for by the data in Figure 8.4, where I showed that correlated control flow in this benchmark causes some inaccuracies in the fabricated flow of control. Overall, however, the fabricated criticality is very precisely reproducing the same average behavior achieved by an online critical path detector and predictor.

8.4.3 Sensitivity analysis

I now explore the extent to which the above good results depend on idealized profiling. There are two dimensions to this. The first is the input sets that drive the profiling runs. I show below that criticality derived from profiles of the SPEC *training* runs, and then used to drive runs of the *reference* input sets, continues to deliver good performance. There are a few cases in which performance is not as good as the dynamic schemes, but this occurs because of a lack of code coverage — an absence of, not inaccuracies within, criticality data. The second dimension is the amount of profile data that must be collected. I will quantify below the extent to which this affects the quality of fabricated criticality.

Program input

Criticality appears to be remarkably stable across different program input sets. I generated profile databases from runs of the SPEC training input sets, and then fed those databases into the trace-fabrication infrastructure. The criticality databases thereby produced were then used to drive precisely the same set of reference input set runs reported on above. The results are shown by the unshaded portions of the bars labeled ‘s’ in Figure 8.9. Performance is, in general, indistinguishable from the self-trained results: most benchmarks show little or no change. Only three — `bzip`, `gzip` and `perl` — stand out as suffering

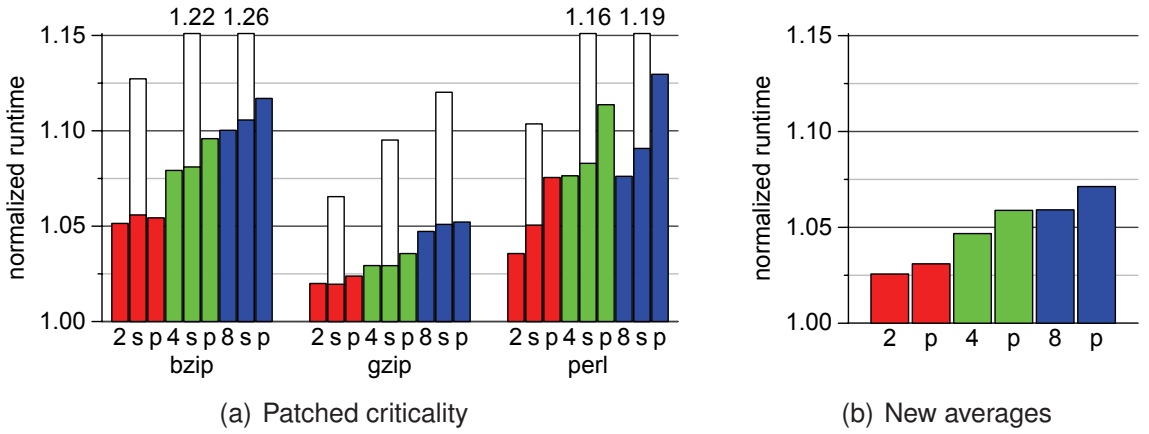


Figure 8.10. Code coverage. The graph on the left shows that performance of the three worst-performing benchmarks can be drastically improved when the criticality database used by the static LoC scheme is supplemented with criticality derived from profiling of the reference input sets (only *missing* information is added; no existing criticality data is modified). The first two bars for each clustered machine repeat the data from Figure 8.9. Bars labeled 'p' show the performance with the patched criticality databases. The graph on the right shows how the overall performance averages change when just these three amended results are factored in.

substantial performance losses. However, those losses can be ascribed to poor code coverage in the training input sets. To demonstrate this, I patched the training criticality database with some of the fabricated LoC derived from the reference inputs. That is, I supplemented the criticality data by adding to it from the self-trained database, but only for instructions for which criticality information is entirely missing; I do not modify any existing data. Figure 8.10(a) shows that the three worst-performing benchmarks now all perform similarly to the rest. With these amendments to just those three benchmarks, the average performance bars from Figure 8.9 change to those shown in Figure 8.10(b) — very close, now, to the idealized results of Figure 8.9.

That code coverage appears to be the only cause for performance loss bodes well for an offline infrastructure in which there is continuous profiling of program execution. In such an environment, the profile databases can be repeatedly updated across multiple runs of an application, steadily improving the ability of the trace fabrication process to achieve complete code coverage.

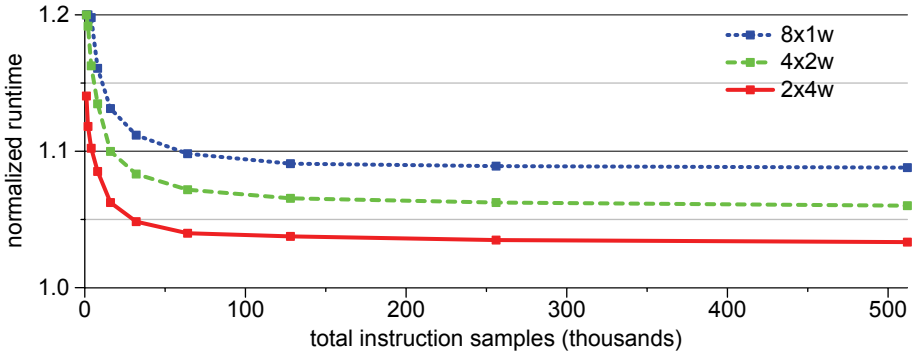


Figure 8.11. Performance as a function of sample count. Each line plots the harmonic mean (across all 12 SPEC Integer benchmarks) of the runtime of each of the 3 clustered machines relative to that of the monolithic machine. The x-axis shows the number of samples in the profile database that drove the trace fabrication scheme.

Quantity of samples

A second factor that was idealized in Section 8.4.2 is profiling of every instruction in the real execution (I assumed a sampling rate of 100%). To explore my sensitivity to this assumption, I could change the sampling rate, but that, by itself, is not a very useful metric: the *time-span* over which sampling occurs can mitigate even a very low sampling rate. A more useful metric, then, is the total number of instruction samples needed before the fabricated traces become sufficiently representative of real execution. To measure this, I adopt an approach in which I distill small, fixed-size profile databases from the fully-sampled databases that I have been using up to this point. This is achieved by randomly picking profile records from the original databases, biasing the choices as per those records' sampled frequency. In a series of experiments, I pick a fixed number of samples — ranging from one thousand to one million — to generate a distilled database, which I then use to drive the trace fabrication logic. I then compare the performance obtainable from each of the resulting criticality databases.

Figure 8.11 shows how the performance of the static scheme improves as a function of the number of profile samples used to drive the trace fabrication infrastructure. Clearly, performance very rapidly converges on its final value, with all machine configurations not showing much improvement beyond 128-thousand samples.

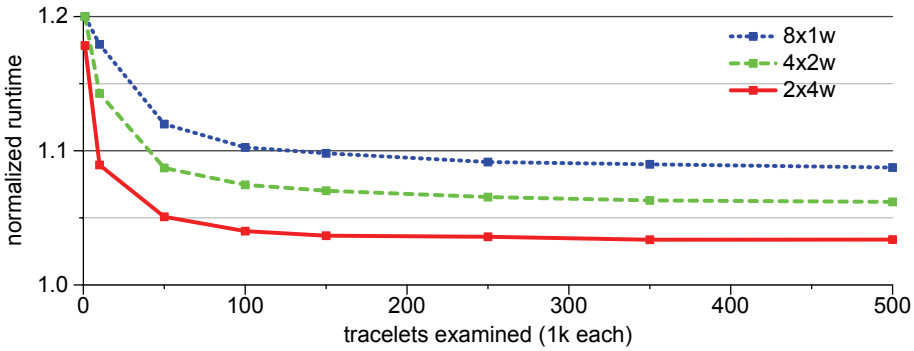


Figure 8.12. Performance as a function of tracelets generated. Like Figure 8.11, each line shows the harmonic mean of runtime on the 3 clustered machines relative to that on the baseline. In these experiments, a complete profile database is used, but the trace fabrication logic is run for a fixed number of 1000-instruction tracelets before a criticality database is emitted.

8.4.4 IPC convergence

I conclude this section by showing that the offline logic tends to converge on its final criticality profiles very quickly. That is, there is no need to fabricate and analyze a very large number of tracelets before a sufficiently accurate criticality database is obtained. Figure 8.12 shows the performance of the three clustered machines relative to the monolithic baseline as a function of the number of tracelets examined before their criticality database is generated. Performance very quickly converges on its final value by the time approximately 100-thousand instructions (*i.e.* 100 tracelets of length 1,000).

8.5 Conclusion

I showed in this chapter that instruction criticality — LoC, in particular — is rather stable, both within and across runs of a program. This is a property that renders it well-suited to offline analysis. I also showed that criticality can be accurately computed using profile information obtainable from hardware already available in commercial processors. My approach is built on two key observations. First, in assessing an instruction's criticality, be it binary or LoC, one is really characterizing what fraction of a static instruction's dynamic instances are critical. Second, this fraction is stable to the extent that one can substitute

representative for real execution traces in order to compute it. I proposed a profile-guided random trace construction method that exploits statistical properties of the program to generate these representative instruction traces, plus an abstract execution method for discovering which store-load pairs in those traces are liable to alias. This approach is extremely effective, achieving upwards of 90% of the benefit of dynamic critical path predictors in hardware.

Although the trace fabrication scheme, as presented, is already modest in terms of its input data and processing requirements, I believe it could be further optimized to reduce the number of tracelets that must be examined before results converge. For example, biasing trace seed selection to cover cold regions of the code, and compensating for this when aggregating the results, could potentially achieve better code coverage with a smaller number of tracelets by removing the need to repeatedly and redundantly explore hot paths.

Profile-guided trace fabrication is an interesting new analysis technique in its own right, one that can potentially serve a number of other applications. It constitutes a compelling design point between traditional trace-driven dynamic analysis and static compiler analysis. One very appealing potential application is the use of fabricated traces as a vehicle for computing a critical path signature for a program, and hence for attributing portions of the observed runtime to factors such as branch mispredictions, cache misses, *etc.* There is considerable demand for such information, as attested by the numerous extant and proposed hardware profiling schemes for computing *CPI stacks*, which apportion average CPI among various performance-degrading factors [29, 63, 92]. The accuracy of these schemes varies with the degree of sophistication engineered into the profiling hardware. Those existing in commercial machines appear to be highly inaccurate, to the point, even, that they yield misleading results. Those proposed in a research context, though more accurate, would introduce non-trivial changes and additions to the microarchitecture [29]. A critical path signature derived from fabricated traces is ideally suited to such applications, for two reasons. First, its apportioning of runtime is based on the critical path, not on

aggregate measures of various events in the pipeline. Its results are therefore liable to be more accurate and more meaningful. Second, since trace fabrication relies only on modest hardware support for profiling, it has minimal ramifications for hardware design; indeed, it can even make use of extant logic. As it stands, however, the infrastructure described in this chapter — the trace analysis component, in particular — is not detailed enough to give accurate CPI breakdowns. For example, accurately attributing runtime lost to the memory system requires a more detailed model than the fixed-latency cache miss penalty I use. I believe such problems are not insurmountable, however. A more sophisticated scheme for modeling a variety of cache miss latencies, for example, might well yield very accurate attribution of time to cache misses. There is clearly room for some interesting further work in this area. I have not explored such avenues simply because they digress from my main objective, which is to move as much as possible of the LoC-based steering and scheduling policy logic into an offline infrastructure.

Chapter 9

Toward a mostly-static dynamic machine

Though removal of critical path detection and prediction logic from hardware, as per the previous chapter, constitutes an important simplification, there remain a number of challenges facing a realistic implementation of the LoC-based steering and scheduling policies developed in Chapter 7. Indeed, the critical path logic is but a small part of the whole, merely providing the input that drives the LoC-based policies. The hardest implementation challenges, which are introduced by the policies themselves, still remain. In fact, one might even argue that those policies, though effective at mitigating partitioned-core performance penalties, effect a zero-sum gain: complexity is removed from the monolithic execution core in partitioning it, but, for the sake of sustaining good performance, must be re-introduced into the machine's front-end in the form of sophisticated steering logic and its attendant structures. I think such a view is too negative, but there is certainly basis in the claim that a hardware implementation of sophisticated steering and scheduling policies somewhat diminishes the appeal of a clustered machine. Complexity-effective design is, after all, the principal motivation for partitioning the core in the first place.

In this chapter, I explore mechanisms for circumventing the hardware complexities associated with LoC-based steering and scheduling. I showed in Chapter 8 that LoC is a comparatively stable property of a program, both within and across runs. It stands to reason, then, that the LoC-based policies, whose decisions are driven by LoC values, will likewise tend to be stable. This does not mean all decisions are statically discernible, but rather that some subset of them will be. My principal aim in this chapter is to explore the extent to which this is indeed the case and, hence, to gauge the potential for developing a scheme in

which a portion of the policy logic is implemented in an offline software component.

Enlisting help from software has a long history in clustered machines. The early VLIW machines, in particular, exposed their clustered microarchitectures in the Instruction Set Architecture (ISA), leaving it to the compiler to perform (static) instruction steering [27, 59]. I mention this because I want to make clear an important distinction between the strategies adopted by the early VLIW machines and those that I will take in this chapter. It is not my goal to shift *all* responsibility for steering onto the shoulders of software and so render the machine completely static. I want to retain in hardware enough dynamic decision making capability to deal effectively with runtime behavior that is not — perhaps cannot be — anticipated statically. At the same time, I want to obviate, where possible, the use of expensive hardware to repeatedly make decisions that could easily have been made statically in software. In short, I seek a judicious division of labor between the static and dynamic parts of the system. In this respect, my underlying philosophy is not unlike that espoused by multipass pipelining [10, 11]. In fact, that work serves as an interesting counterpoint to my own: whereas the aforementioned scheme introduces some degree of dynamic behavior into an in-order, software-managed pipeline, I propose displacing into software some of the statically-obvious work currently performed by hardware in an out-of-order machine.

Section 9.2 introduces the hardware/software framework in which I pursue this work. Broadly, this is a co-designed virtual machine infrastructure [91] in which microarchitecture and ISA are simultaneously and cooperatively designed, each aiming to provide whatever performance-enhancing features might benefit the other. At the same time, an ISA distinct from that implemented by the hardware is made visible to software. This, being the medium in which all programs are expressed, remains fixed irrespective of (performance-driven) changes to the underlying microarchitecture and its closely allied ISA. In line with previous work, I refer to the ISA implemented by the hardware as the *Implementation ISA* (I-ISA) and to that made visible to software as the *Virtual ISA* (V-ISA) [3]. Such a machine

is virtual in the sense that the ISA in which all programs are expressed is not understood by hardware: a layer of software is required to translate V- into I-ISA code. Most of my focus in this chapter falls on that layer of software, since it is there that the offline component of the LoC-based policies is implemented.

I want to emphasize that the co-designed virtual machine is not itself the object of study in this chapter. I view it merely as a convenient framework in which to explore software-assisted implementation of steering and scheduling policies. It is ideally suited to that end because it permits freely exposing all the details of the clustered microarchitecture in the I-ISA, thereby making them visible to the software responsible for making steering decisions. Equally, any performance-enhancing requirements imposed by that software component can easily be communicated, via the customized I-ISA, to the hardware, or even engineered directly into the hardware; in neither case is the result visible to application software. In short, a co-designed framework admits a degree of coupling between hardware and software that is not generally possible when that coupling, which is liable to change across processor generations, is simultaneously visible to application and operating system software. Though I rely on these benefits, they are not details I will focus on. It is not my objective to define either of the two instruction sets in any detail, nor is it to formulate a mode of operation for the virtual machine infrastructure — that would constitute a large study in its own right. Instead, my goal is to quantify the *potential* of static decision making and, more specifically, to develop algorithms for making those static decisions effectively. I regard these issues as a necessary first step: there is no point pursuing the details of a co-designed framework if it turns out static decision making has only limited potential. My objective, then, is to lay the groundwork in this new area and, hopefully, thereby stimulate further exploration.

I describe the software component of the co-designed infrastructure in Section 9.3. Its task is to make static steering decisions, thereby removing some — but not all — of the burden from hardware. I retain dynamic steering logic, but it is now reduced to a compar-

atively simple dependence-based policy, extended to support software-directed proactive stalling and load-balancing behavior. This offline steering logic is centered on a technique I call *instruction aggregation*, which involves grouping RISC-like *micro-ops* (μ-ops) into larger entities called *macro-ops* (m-ops). The latter are the unit at which instruction steering occurs: dynamic steering logic makes one decision for each m-op, thereby implicitly steering all constituent μ-ops to the same cluster. In addition to serving as a means for encoding steering decisions, the aggregation brings a number of *amplification* benefits to the hardware. In particular, steering logic, having now to make decisions at the coarser granularity of m-ops, can make fewer decisions per cycle while sustaining the same effective μ-op throughput. I describe these and other benefits in more detail in Section 9.3.

Section 9.4 briefly discusses the hardware components of the co-designed infrastructure. My emphasis there is on LoC-based dynamic scheduling. This turns out to be the least amenable to assistance from software, demanding that hardware remain capable of distinguishing individual instructions based on their LoC value. As I showed in Chapter 8, however, static LoC values suffice for this purpose. Moreover, m-ops serve as an ideal means via which to convey those static values, *en masse*, to the hardware.

I present the results of a preliminary evaluation of the co-designed system in Section 9.5. I say *preliminary* because the results, though very promising in terms of performance, do not deliver enough of the amplification benefits to which I alluded earlier. The principal cause for this is *singletons* (m-ops comprising just one μ-op), which account for about 45% of all m-ops. To some extent, the problem is an artifact of the intra-block constraint to which I subject the aggregation logic. However, many of the singletons arise because of some naïvety in the aggregation logic. Section 9.6 presents a number of enhancements to the basic algorithm, which, together, largely eliminate the singleton problem.

Overall, the results I present in this chapter bode well for the prospects of a combined hardware/software solution for clustered machines. The instruction aggregation scheme I

have developed is able to deliver almost the same performance as a fully dynamic infrastructure, and at the same time demands only very modest assistance from hardware. This is just a first step, however. I conclude this chapter with a discussion of directions for future work in this area. I describe there a more comprehensive framework in which m-ops are promoted to first-class entities in the I-ISA, becoming now instructions in their own right, not just meta-data used by steering logic. A microarchitecture that processes these macro-ops in its front-end is very appealing in terms of its potential to bring further amplification benefits to the front-end, particularly to register rename logic.

9.1 Implementation challenges

As I noted earlier, after critical path detection and prediction logic is removed from hardware, there remain a number of challenges facing a realistic implementation of the LoC-based policies. I briefly discuss the most important of those challenges below.

9.1.1 Dynamic steering

The proposed stall-versus-steer and proactive load-balance schemes are both enhancements of a previously published *dependence-based steering with load-balancing* scheme [49]. Broadly, this operates in three steps. In the first, a state table is interrogated to determine the current assignment of in-flight value-producing instructions to clusters. The second step involves using instruction dependence information to assign, where possible, each consumer to the cluster at which its producer resides. If that cluster is full, or if the producer has already issued, the load-balance component is activated to select the least busy cluster. Finally, the resulting steering decisions are written back to the state table for use in subsequent cycles. Viewed in this light, dynamic steering is very similar to register rename logic; and it suffers from the same scalability problems. Most problematic is providing single-cycle access to the the state table [76], which requires 2 read ports and 1 write port

for each instruction steered per cycle. A second problem is posed by intra-cycle dependences, which require complex circuitry to ensure that a steering decision made for one instruction is visible to, and can thus affect the decisions for, dependent instructions that are steered in the same cycle.

Though an LoC-driven stall-over-steer refinement of the above behavior is easy to implement, the proactive load-balancing requirement is more problematic. It must somehow be able to tell, in advance, that a given instruction, which would otherwise successfully collocate with its producer, should rather be steered away because more important consumers are yet to be seen in the instruction stream. Achieving this end demands that steering be further embellished with a predictor to identify the load-balance candidates. That predictor must be trained by new back-end logic that detects cases in which an instruction is not the most critical consumer of its register operands.

9.1.2 Dynamic scheduling

An LoC-based scheduling policy gives priority to ready instructions based on their associated LoC value. This implies a requirement to prioritize based on a numerical value associated with each instruction. Conventional dynamic schedulers, however, use a relatively simple technique to statically assign priority based on an instruction's position in the window [76]. However, there are schemes that can prioritize based on numerically encoded priorities [17], and these are quite likely to be able to meet the very strict timing constraints to which select logic is subject. (I describe this in more detail in Section 9.4.) That said, the efficacy of LoC-based scheduling is very sensitive to accurate criticality information being available on a per-instruction basis. Since I am removing the critical path detection and prediction logic from the hardware, there is now a requirement to associate LoC values with individual instructions, and to communicate those values to the hardware. My evaluation of static LoC in Chapter 8 (necessarily) ignored the implications of this requirement, but now a practical solution is needed.

9.2 Co-design

It is clear from the above discussion that a hardware-only implementation of the LoC-based policies would burden the pipeline with a number of supporting structures and, in the case of steering itself, would pose a scaling problem analogous to that already faced by register rename. Though it would, in principle, be possible to implement each of those components, doing so would surely counter the complexity benefits that motivated partitioning the execution core in the first place. The central thrust of this chapter is that most of that hardware is either superfluous, in the sense that its work can be performed with equal effectiveness statically, or over-provisioned, because a simpler version could be used if appropriate assistance from software were available. In this section, I describe the rudiments of a *combined hardware-software scheme* that I will use in this chapter to justify that claim.

9.2.1 Assistance from software

My objective in this chapter is to move some fraction (not all) of the steering decisions offline. Included in that objective is complete removal from hardware of the proactive load balancing scheme, the most demanding in terms of its hardware requirements, yet, given its inherent dependence on knowledge of future dataflow, also the most natural candidate for offline implementation. I have already alluded to the principle that motivates this approach: a good fraction of dynamic steering decisions will likely be stable, simply because the LoC values that drive those decisions are themselves stable. I exploit this fact by performing *instruction aggregation*, a technique in which I use intra-block dataflow dependences, plus knowledge of per-instruction LoC, to group RISC-like *micro-ops* (μ-ops) into larger entities called *macro-ops* (m-ops). Accommodating this offline component is the new microarchitecture depicted in Figure 9.1.

The m-ops constitute what I call *collocation groups*, since the constituent μ-ops are constrained to be collocated at the same cluster. Enforcing this rule is a modified dependence-

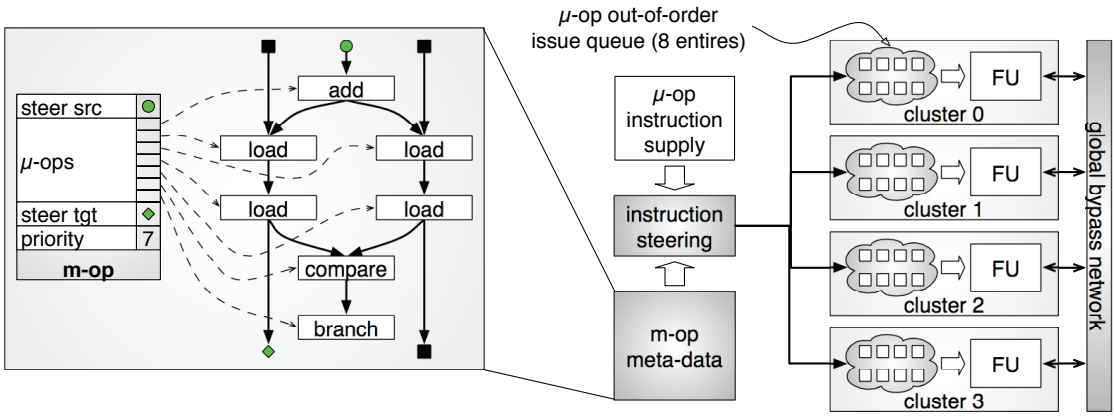


Figure 9.1. A microarchitecture for instruction aggregation. Instruction steering now operates at a coarser granularity, making one steering decision for each macro-op (m-op) seen in the instruction stream. However, it is still μ-ops that flow through the front-end pipeline; identification of m-ops is by means of "macro-op meta data" embedded in the I-ISA binary. The details of how that information is encoded in the binary are beyond the scope of this preliminary study.

based dynamic steering unit, whose responsibility is to decide where each m-op — and hence all constituent μ-ops — will be sent. As Figure 9.1 shows, m-ops are tagged (by the offline logic) with a single register source and target specifier to guide the dependence-based steering logic. Thus, the overall task of steering is divided between offline logic, which forms the collocation groups, and online logic, which dynamically picks clusters to which each collocation group will be sent. The μ-ops remain, as usual, the unit at which execution occurs. LoC-based dynamic scheduling is still needed for this purpose, but the relevant priority information is communicated to the scheduler via a single LoC value associated with each m-op (see Figure 9.1). All the μ-ops within a given m-op inherit that LoC value.

Using aggregation as the means via which static steering decisions are communicated to the hardware is appealing in at least two respects.

1. *Steering complexity.* The hardest parts of the steering policies can now be implemented offline. By aggregating instructions into m-ops, software decides which instructions should be collocated. Equally, by choosing not to collocate them, software makes proactive load balancing decisions. And because software tags each m-op

with input and output register specifiers, it indicates to the steering logic the producers toward which a given m-op ought to be steered. That is, software decides which are the important producers to follow. Finally, software can tag important m-ops as candidates for stall-versus-steer behavior; hardware need only respond accordingly when the preferred steering decision is not possible.

2. *Bandwidth amplification.* Because decisions are now made at the coarser granularity of m-ops, dynamic steering logic benefits from a *bandwidth amplification* effect, permitting it to support fewer decisions per cycle while sustaining the same effective throughput into the execution core.

Amplification effects can also benefit other parts of the microarchitecture, particularly the front-end. This is a subject to which I will return in Section 9.7.

As it stands, my description of the co-designed infrastructure is vague on a number of points. In particular, the details of how m-op information is encoded in the instruction stream — the m-op "meta-data" in Figure 9.1 — have not yet been specified. I deliberately leave those unspecified at this early stage: I regard a proper analysis of such aspects of the system as contingent upon first answering some more basic questions about potential.

1. How does aggregation work? — what algorithms are needed, and what profiling infrastructure (if any) is needed to make the right collocation decisions?

2. What is the cost to IPC of migrating some of the decision making into an offline component? That is, to what extent is decision making on a per-dynamic instruction basis really necessary?

3. How much of an amplification effect can steering logic benefit from?

In Sections 9.3 through 9.5, I present the details of and results from a preliminary study aimed at tackling these questions. Therein, I confine the analysis to a scheme in which instruction aggregation is restricted to the scope of a single basic block (*i.e.* m-ops cannot span blocks).

9.2.2 Related work

The instruction aggregation process I described above is an example of a broader set of techniques that share, as their main objective, the *amplification* of the bandwidth or the effective capacity of various pipeline structures. Recent studies that tackle register rename, result bypass, issue logic and register file size typify this kind of work [14,15,42,54,87,88]. The Intel Pentium M processor also uses a form of aggregation, but in this case primarily to save power [38]. I draw attention to this work because it stands in contrast to my own. Although amplification is a benefit I hope to exploit, it is not a metric I aim to optimize. I use aggregation, first and foremost, as a means for capturing static steering decisions. Recent work by Kim and Smith [52, 53] is, in this respect, more closely related to my own. They explore a form of instruction aggregation to guide a dynamic dependence-based steering scheme. However, their target is a laned machine, and they adopt a simple accumulator-oriented (I-)ISA to encode the dependence chains that will be steered to the same lane. While an accumulator-oriented ISA is indeed appealing, I have argued already that a laned machine with a dependence-based steering scheme will inevitably incur substantial IPC losses.

9.3 The software component

Broadly, the task of the offline software component is to determine which μ-ops will benefit from being collocated at the same cluster, and hence which μ-ops ought to be aggregated into the same m-op. Its overall infrastructure is similar to Digital's *FX!32* dynamic binary translator [21], in the sense that it operates in two main steps: a profiling phase followed by off-line analysis and code generation. However, whereas *FX!32* uses emulation to gather the profile information it needs, I make use of a scheme very similar to the profile-guided trace fabrication infrastructure presented in Chapter 8. Figure 9.2 depicts its high-level organization.

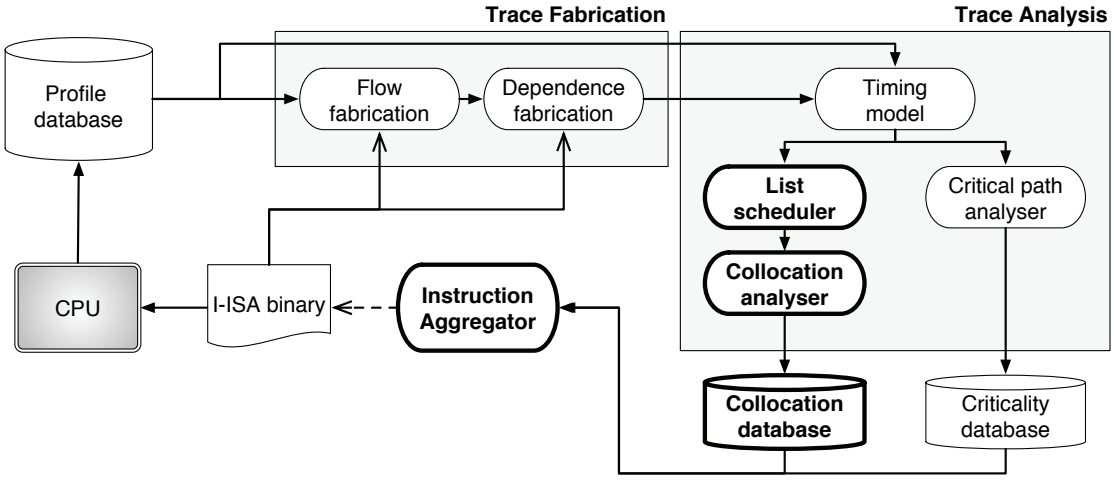


Figure 9.2. Offline components of a co-designed infrastructure. (*cf.* Figure 8.2, page 150.) The infrastructure is built on the trace fabrication framework described in Chapter 8. Additions to that framework are highlighted in this diagram.

As before, I divide the logic into a trace fabrication and a trace analysis component. The first of these is no different from the scheme I described in Chapter 8 (Section 8.2, page 153), comprising a profile-guided random walk through the program binary to fabricate the flow of control, together with an abstract execution component to fabricate memory dependences. By contrast, the second component — trace analysis — is now very different from the comparatively simple critical path analysis presented earlier (Section 8.3). As Figure 9.2 shows (highlighted portions), the trace analysis component has been extended with a *collocation analysis* part, the principal task of which is to identify those instructions that are good candidates for aggregation into a single m-op. The output of this process is a *collocation ratio* (CR) database, which, together with an LoC database (produced as per Section 8.3), serves as input to the final step in the offline component, *instruction aggregation*. The following two sub-sections describe, respectively, collocation analysis and instruction aggregation in more detail.

9.3.1 Collocation analysis

As I noted above, the objective of collocation analysis is to discover which groups of dependent instructions will benefit from being steered to the same cluster. Analysis of instruction traces is ideal for this purpose because traces naturally capture dominant control flow patterns and simultaneously provide a global context for dataflow in hot basic blocks. Both of these factors must be taken into account if intelligent intra-block aggregation decisions are to be made.

Collocation analysis comprises two parts. The first involves list scheduling of the timestamp-annotated instruction tracelets emitted by the timing model (see Figure 9.2). This process is exactly analogous to the idealized list scheduling study I used in Chapter 6 to quantify the underlying performance potential of clustered machines. As before, the scheduler partitions each tracelet into contiguous regions, and then uses oracular knowledge of the whole region's dataflow to prioritize its slotting of instructions. It is configured to slot instructions to the same clustered microarchitecture to which the I-ISA binary is targeted. The schedules it produces constitute the ideal in terms of instruction placement for each (fabricated) instruction trace and, as such, embed a wealth of information on which instructions ought to be collocated with one another.

Gleaning that information from the resulting schedules is a task that falls to the *collocation analyzer*. It simply analyzes the schedule for each region, recording, for each static instruction, the following:

1. **Collocation Ratio (CR)**: the fraction of instances that the scheduler opted to collocate with a producer.

2. **Favored Operand**: the architected register that most frequently induces those collocations.

I say that an instruction *collocated* with a producer if it was slotted by the scheduler, not only at the same cluster as the producer, but also before that producer's value became

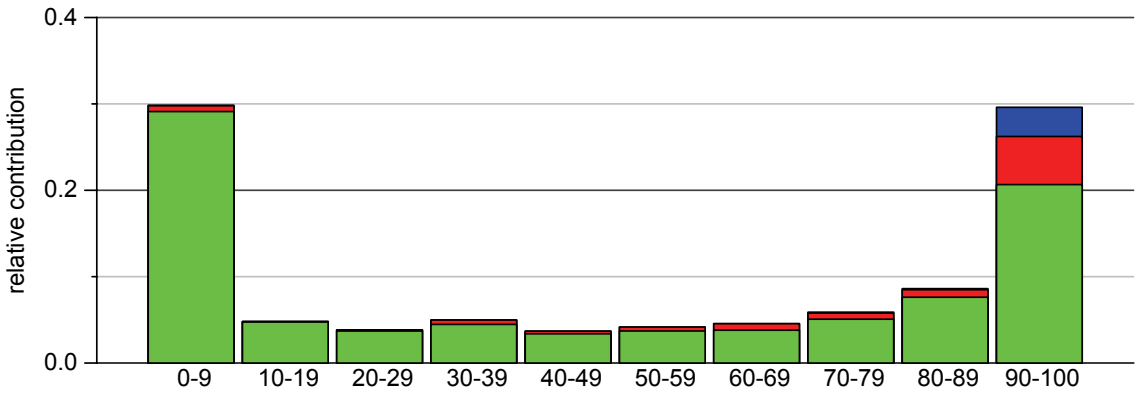


Figure 9.3. Distribution of collocation ratios. Data is averaged across all 12 Spec Integer benchmarks, with each static instruction's CR value weighted by its dynamic count. The different shades in each bar denote the contribution to each CR bucket of various degrees of LoC: darker regions represent higher LoC values.

globally available.

A summary of CR data for the 12 SPEC Integer benchmarks is plotted in Figure 9.3. The distribution is quite strongly bimodal, so the bulk of instructions almost always collocate, or they almost never do.[1] Together, these two scenarios account for 60-80% of all dynamic instruction instances; the remaining 20-40% fall into the middle region of the CR distribution where the tendency to collocate is not very biased. That the CR distribution is so stable is not very surprising. The list scheduler deliberately gives priority to, and hence tends to collocate, important dataflow chains. These, in turn, tend to be quite stable because they derive either from the longest dataflow paths through the program (*e.g.* chains of induction variable updates that span loop iterations), or from the backward dataflow slices of mispredicting branches and loads that miss in the cache. In both cases, these are largely static properties of the program.

It is also clear from Figure 9.3 that all high-LoC instructions exhibit high CR values. This is not surprising given that the list scheduler tends to collocate important dataflow chains. However, the converse is not true: not all instructions with high CR values have high LoC, as is evidenced by the strong contribution from low-LoC instructions in the

[1]Since most instructions are monadic, the favored operand trend is even more strong: over 90% of all collocations are induced by a single operand.

Algorithm 2 Instruction Aggregation

```
 1: MOps ← ∅
 2: Producers ← ∅
 3:
 4: for all μ-op ∈ BasicBlock do
 5:    src ← FavoredOperand( μ-op )
 6:    m-op ← Producers[ src ]
 7:    if m-op = λ or CR(μ-op) ≤ LoadBalanceThreshold then
 8:       m-op ← MakeMacroOp( μ-op )
 9:       MOps ← MOps ∪ m-op
10:    else
11:       AddToMacroOp( m-op, μ-op )
12:    end if
13:    m-op.LoC ← max( m-op.LoC, μ-op.LoC )
14:    if m-op.LoC ≥ StallThreshold then
15:       m-op.StallFlag ← True
16:    end if
17:    t ← TargetRegister( μ-op )
18:    Producers[ t ] ← m-op
19: end for
20:
21: return MOps
```

rightmost bars in the figure. Thus, the tendency to collocate is not exclusive to very critical instructions; even unimportant dataflow tends to get collocated. From an instruction aggregation point of view, therefore, it is clear that effective cluster assignment demands grouping of instructions based on CR values, not just on LoC values.

9.3.2 Instruction aggregation

Algorithm 2 summarizes the intra-block aggregation logic. In a forward pass over each block, a set of partially formed m-ops is maintained, together with a map to record which m-op currently produces the latest version of each architected register. At each step, the logic either adds to an existing m-op (line 11) or creates and adds a new one to the set (lines 8–9). The decision to add to an existing m-op or form a new one (line 7) is determined by two factors: whether the μ-op's preferred operand is produced locally and,

if so, whether its CR value is sufficiently high to make joining the corresponding m-op an attractive option. Exactly what constitutes *sufficiently high* can be determined empirically. Different benchmarks benefit from different values, but I find that fixing the value at 0.4 produces good overall results.

Also shown in Algorithm 2 is the static component of the stall-versus-steer policy. I assign an LoC value to each m-op by computing the maximum of its constituent μ-ops' LoC values (line 13). If this value exceeds a threshold, the m-op is flagged as a stall-over-steer candidate (lines 14–16); dynamic steering will stall the front-end if the cluster to which it would like to steer that m-op is currently full. Once again, the stall threshold is determined empirically.

The results of applying this aggregation logic to a hot loop from `vpr` are shown in Figure 9.4. This is the same code explored earlier in Chapter 7 (Figure 7.4, page 129). The loop body contains a chain of highly critical dataflow (LoC is above 90%) with several less important chains that feed off it. I draw attention to instruction `L`, which forms part of the critical chain: of its three consumers, only `N` resides on the critical chain. In this case, `N`'s high CR value results in its inclusion into the m-op headed by instruction `A`; instructions `M` and `O` are *proactively* pushed away from their producer because their CR values are below the inclusion threshold. This decision is exactly the right thing to do in this loop because it permits successive dynamic instances of the critical chain to collocate without having to contend for issue slots with the unimportant instructions `M` and `O`, nor with the chains of (unimportant) dataflow that emanate from them. I want to emphasize that this decision would be hard to make in a dynamic context because of the fetch-order constraint: instruction `M` is seen before instruction `N`, yet the latter is the preferred collocator, not the former. That the CR metric performs so well here is a result of its *global* nature. It reflects the tendency of the list scheduler, which has full knowledge of proximate dataflow, to give precedence to successive instances of the important chain.

Instruction `S`, a dyadic consumer, demonstrates the utility of the favored operand metric.

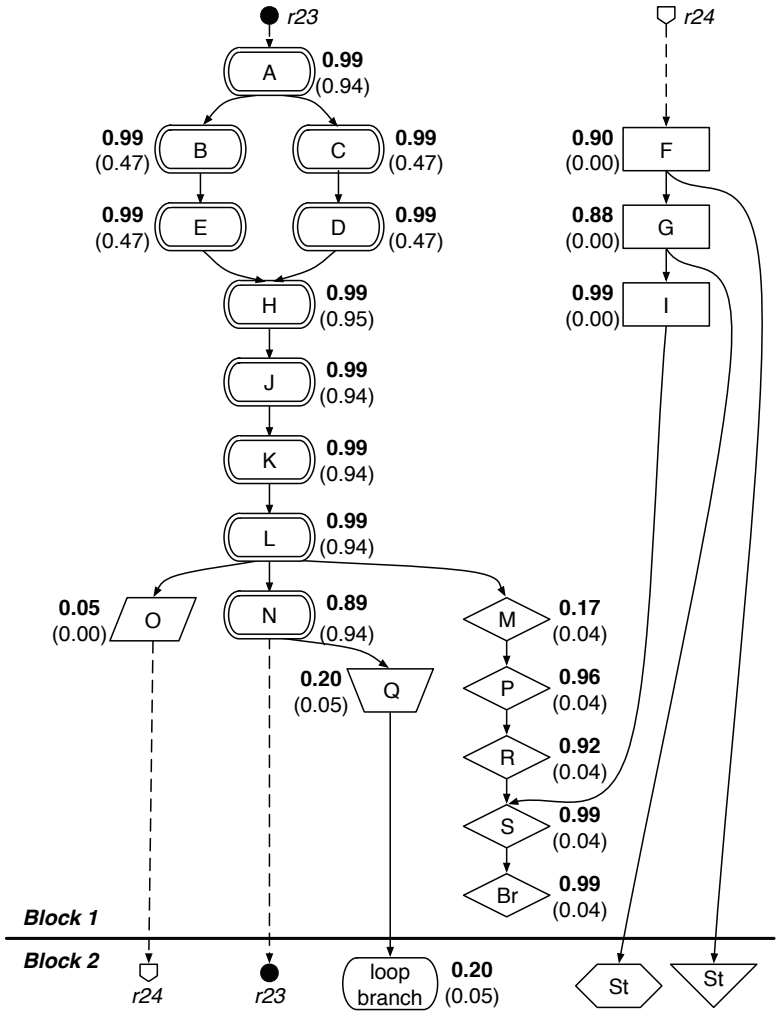


Figure 9.4. Aggregation example. Dataflow in a hot loop in `vpr`. The loop body comprises two basic blocks, the first being terminated with an early exit check. Instructions are labeled in their fetch order and are annotated with their LoC (bottom, parenthesized) and CR (top, bold) values. Different shapes denote m-op membership (there are 5 m-ops in Block 1, for example). Loop-carried dependences are indicated by the dashed lines: `N` feeds `A` and `O` feeds `F`.

The aggregation logic chooses to collocate it with `R` because it is via the `R`–`S` dataflow edge that most of its collocations were induced. That trend, in turn, arises because the list scheduler attempts to collocate a consumer with the producer of its last-arriving — that is, its more critical — operand.

9.3.3 Macro-level dependences

Recall from Figure 9.1 that the co-designed machine still performs dynamic steering, but that this now operates at the level of m-ops. Since the steering policy I adopt is dependence-based, it remains for me to define m-op input and output specifiers upon which the steering logic can base its dependence-based decisions.

From an input point of view, a m-op can consume externally-defined values at any one of its constituent μ-ops, but, from a steering point of view, it ought only to expose the critical dependences to the steering logic. To do so, I exploit the fact that m-ops are, by construction, chains of dependent instructions whose constituent μ-ops get their most critical operand from *within* the chain (excepting, of course, the head instruction). More succinctly, the most critical external value will not arrive as a side-input.[2] I therefore use just the head μ-op in each m-op to specify input dependences at the macro-level. If that μ-op's CR value is sufficiently high, I use its favored operand; otherwise, I deem the m-op unimportant and omit an input specifier, flagging it thus for load-balance.

Returning to the example in Figure 9.4, this means the m-op holding the critical dataflow chain (headed by instruction `A`) publishes `r23` as its input specifier. It so happens that `r23` is (usually) produced by instruction `N`, so this m-op will tend to collocate with itself. Likewise, the m-op headed by `F` publishes `r24`, so it will be steered toward the singleton m-op containing `O`. All other m-ops in the block are headed by μ-ops with low CR, so they do not publish any input dependences and will accordingly be load-balanced.

Note that because I specify just a single input dependence on each m-op, the steering logic state table need only supply one read port for each m-op it steers. This is half the bandwidth required of a table for a conventional, dyadic ISA. Bearing in mind that the total number of m-ops is smaller than the original μ-op count, the total reduction in read bandwidth is liable to be quite significant.

[2] If this were not the case, then an internal μ-op would receive its preferred operand from outside the m-op, a scenario precluded by Algorithm 2.

My policy for choosing m-op output specifiers has already been alluded to: I use the target register of the last value-producing μ-op. Once again, this approach exploits the fact that m-ops comprise chains of dependent instructions, but in this case I also rely on the tendency of CR-based aggregation to group like-with-like in terms of LoC. As Figure 9.4 shows, m-ops are homogeneous in terms of their constituent μ-ops' LoC values. It will therefore seldom happen that an internal μ-op produces a value that is needed more urgently outside the m-op than it is inside; publishing the last-produced value is therefore not going to prevent important consumers from following the m-op. Instruction `L` exhibits exactly this property.

9.4 The hardware components

There are two main parts to the hardware component of the LoC-based policies. The first is the dynamic steering logic. The previous section's discussion of register input and output specifiers has already alluded to the operation of the dynamic steering logic. My discussion below is therefore brief. The second component is the dynamic scheduling logic. This is the part of the LoC-based policies that is least amenable to assistance from offline logic: I find that good performance relies on the ability to make fine-grained scheduling decisions at runtime.

Steering

Instruction steering, which operates at the granularity of m-ops, implements a very basic dependence-based policy, modified now to stall when collocation of m-ops that have been flagged as stall candidates (recall Algorithm 2, line 15) is not possible. The proactive load-balancing requirement is also taken care of by instruction aggregation because consumers that ought not to collocate with a producer (because they are not sufficiently important) are not aggregated with that producer (if it is a local one) or are not flagged to follow

that producer (if it is remote). The dependence-based logic therefore has a very simple task: collocate m-ops as per their source register specifier; load-balance if that specifier is absent, or if collocation is not possible and the m-op is not flagged as a stall candidate.

Scheduling

As I noted above, the scheduler offers the least opportunity for migrating functionality offline. In my evaluation of a number of different schemes, I found that performance is very sensitive, not only to having LoC information available in the scheduler, but also to having it available at a per-instruction (μ-op) granularity. Thus, removing the on-line LoC infrastructure from the pipeline imposes the rather difficult requirement that LoC information still be made available for each μ-op in the window. Doing so at the granularity of μ-ops would impose a substantial encoding burden — an additional 3 or 4 bits to encode sufficiently many degrees of LoC.[3] However, instruction aggregation logic can help here. As I noted already, aggregation tends to group μ-ops with similar LoC together, so, without much loss of information, LoC can be communicated at the m-op level. I therefore tag each m-op with the largest of its μ-ops' LoC values (Algorithm 2, line 13). All constituent μ-ops inherit that value when they are dispatched to their cluster's issue queue, thus making per-instruction LoC values available to the scheduler.

The scheduler itself requires modification. I stated earlier that conventional scheduling logic, which uses an instruction's position within the window to determine its priority, cannot be adapted to prioritize based on numerical LoC values. However, an alternative scheme, which is depicted in Figure 9.5, was developed by Buyuktosunoglu *et al.* for the purposes of implementing an oldest-first selection policy in a non-collapsing instruction queue [17]. Briefly, the logic works as follows. A 3-bit encoding of priority, which is associated with each window entry, is fed into a simple circuit that checks if an entry's request should be granted. This check amounts to a sequential filtering process, from high-

[3] I ignored this issue in my evaluation in Chapter 8 of the static LoC schemes. Catering to the practicalities of having to statically encode LoC would have detracted from the main focus of that study.

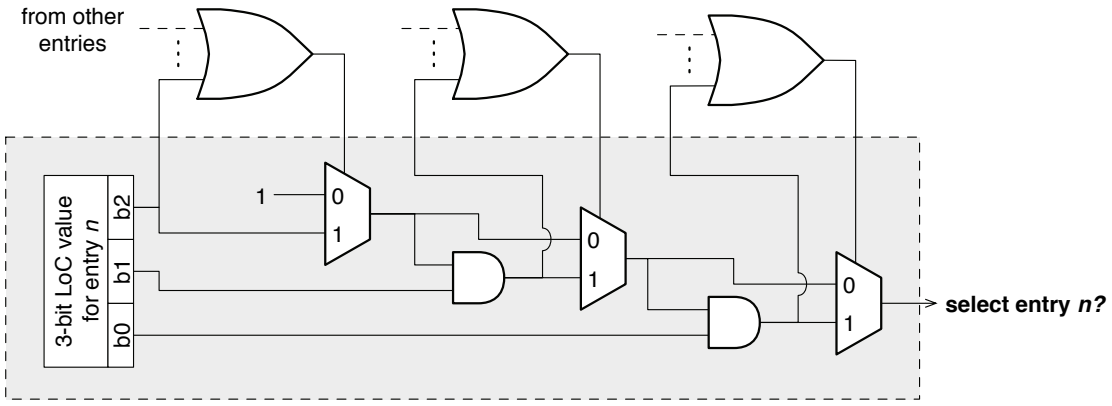


Figure 9.5. LoC-based select logic. The shaded region shows the logic for a single window entry; the OR-gates are not replicated.

to low-priority bits, where each window entry checks if another entry has a 1 in an encoding bit while its own bit is 0. If such an entry exists, this entry's request can immediately be denied; if not, the lower priority bits are likewise checked. Any entries whose select line is high are therefore guaranteed to have priority that is not lower than any others. Ties can be resolved by a simple static scheme that selects from among the winning entries.

9.5 Evaluation

Continuing with the empirical framework I used in the previous two chapters, I evaluate three configurations of clustered machines, each a different partitioning of an 8-wide monolithic out-of-order superscalar machine. Those machines, which I will again refer to as the 2x4w, 4x2w and 8x1w configurations, are configured, as before, to partition the execution resources of an 8-wide monolithic machine (see Table 5.1, page 110). To gauge the efficacy of the offline logic described in this chapter, I compare the performance of the co-designed clustered machines to the fully-dynamic — and somewhat idealized — configurations evaluated in Chapter 7 (Section 7.5). I will refer to the latter as the *online* configurations and, by way of contrast, to the co-designed machines as *offline*.

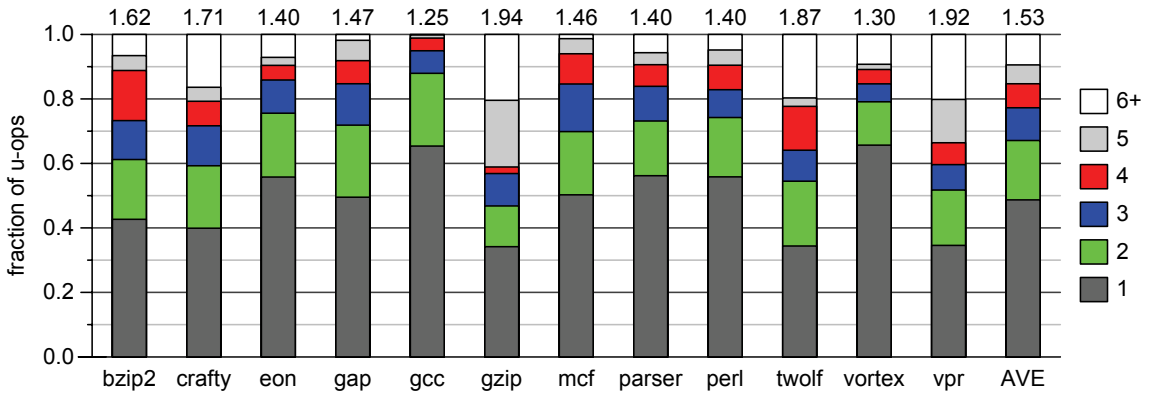


Figure 9.6. Distribution of m-op sizes. Data is for the 4x2w machine configuration; the other machines show similar distributions. The number at the top of each bar shows the average m-op size.

9.5.1 Instruction aggregation

I begin the evaluation with a brief characterization of the m-ops formed by the aggregation logic. I use m-op size for this purpose, since it is a metric that gives a good overall impression of the extent to which aggregation is able to successfully collocate instructions.

Figure 9.6 shows the m-op size distribution for each of the SPEC Integer benchmarks. On average, aggregation logic manages to statically collocate about 1.5 μ-ops. This means steering logic will, on average, make about two-thirds the number of dynamic steering decisions in the co-designed infrastructure as it does in the fully dynamic machine. Though this does represent an improvement, an average m-op size below 2 is, from an amplification point of view, somewhat disappointing. The principal cause for this low number is the preponderance of *singletons* (m-ops comprising just one μ-op), which account for about 50% of all m-ops. If I exclude those singletons from the breakdown in Figure 9.6, average m-op size increases to over 3, indicating that collocation is very effective when it does occur. Though amplification is not the primary objective in aggregating instructions (obtaining good collocations is), a better ratio of m-ops to μ-ops is certainly worth pursuing. Section 9.6 is devoted to that subject.

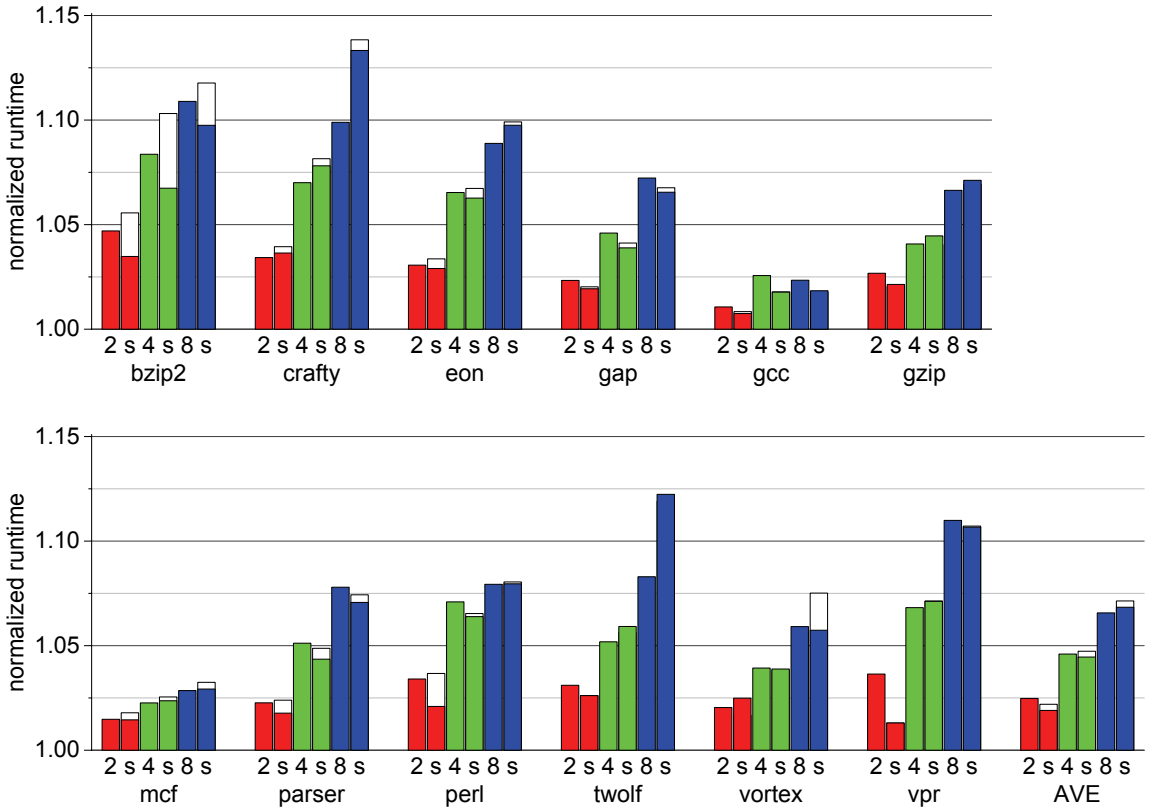


Figure 9.7. Performance of combined hardware-software schemes. Bars labeled '2', '4' and '8' show the normalized runtime of the online (dynamic) LoC-based policies on the 2-, 4- and 8-cluster machines, respectively. The adjacent bars, labeled 's', show the runtime of the corresponding offline configurations. The shaded portions of those bars show the results of driving the aggregation logic with LoC and CR values derived from real instruction traces; the unshaded portion (not always visible), to LoC and CR values derived from fabricated instruction traces. All bars are normalized to runtime on the monolithic machine.

9.5.2 Performance

Figure 9.7 compares the performance of the offline schemes to their online counterparts. So as to distinguish the artifacts of trace fabrication from those of static decision making, performance of the co-designed machines (bars labeled 's') show two different runtimes for each clustered machine configuration. The first, shown by the shaded portion of each bar, corresponds to an idealized setup in which the aggregation logic uses CR and LoC data derived from real instruction traces (the same traces subsequently simulated). The second, which is given by the unshaded portion of each bar, is the runtime achieved when aggregation is based on CR and LoC derived from fabricated instruction traces — as per

the aggregation framework depicted in Figure 9.2. That the latter result seldom exceeds the former is further confirmation of the ability of fabricated traces to very accurately reflect real execution. The more important result, however, is that performance of the offline scheme is, on average, indistinguishable from that of the online one. In fact, aggregation outperforms the online policies on benchmarks like `bzip2`, `gap` and `perl`. Such cases arise because CR values, which drive the aggregation decisions, reflect the tendency of an idealized list scheduler to collocate the most important dataflow chains; the online policies are not as effective simply because they do not have the same global view of dataflow that is available to an offline component. Only two benchmarks stand out as performing significantly worse than the online schemes: `crafty` and `twolf` on the 8x1w machine. In both cases, the predominant cause for the performance loss is dispatch stalls introduced by instruction steering. This problem arises when μ-ops belonging to large, but non-critical, m-ops cannot be dispatched because the cluster to which the m-op has been assigned is currently full. Whereas the dynamic policy would simply load-balance these low-LoC μ-ops in such cases, the static scheme, being constrained to send all constituent μ-ops to the same cluster, must stall when that cluster is full. This problem can possibly be mitigated by fine-tuning the aggregation logic to cap the sizes of unimportant m-ops, thereby reducing the chances that low-LoC instructions induce dispatch stalls. However, since this is not a very significant problem, and since it occurs only on the 8x1w machine, a detailed exploration of solutions is not, at this point, warranted.

Regarding the collocation threshold value, I should note that the data presented in Figure 9.7 is obtained using a fixed threshold value for all benchmarks: a μ-op must have a CR value of at least 0.4 before it is permitted to join its producer's m-op (recall Algorithm 2, line 7). Likewise, a fixed LoC threshold value of 0.3 determines whether a given m-op is flagged as a stall-over-steer candidate (Algorithm 2, lines 14–16). I selected both these values based on brute-force exploration of various combinations. Although I do not have a mechanized way for deriving such parameters, having a practicable scheme for doing so

would open up potential for tailoring their values based on application and target machine specifics. This might be useful for cases like `crafty` and `twolf` on the 8x1w configuration, where overly-aggressive aggregation is resulting in low-LoC m-ops becoming unprofitably large.

9.5.3 Collocation efficacy

To confirm the utility of the CR metric as a means for guiding collocation, I conducted a set of experiments in which instruction aggregation foregoes the list scheduling analysis described in Section 9.3. My objective in doing so is to demonstrate two key points. First, a purely dependence-based scheme that ignores the relative criticality of producer and consumer will incur contention stalls because too much work will be collocated. Second, any strategy that attempts to proactively push consumers away from their producers without taking criticality into account will introduce spurious intercluster communication onto the critical path.

I can demonstrate the first point easily by setting to 0 the proactive load-balance threshold (Algorithm 2, line 7), thereby forcing all intra-block consumers to be absorbed into their producer's m-op. To validate the second point, I implemented a policy that permits only the first sequential consumer of a value to collocate with its producer; subsequent consumers are proactively pushed away. This approach, which constitutes a form of *greedy* collocation, is akin to the dependence-based steering policy from Palacharla *et al.* [76], which I explored at length in Chapter 3, as well as to the scheme used by Kim and Smith in their ILDP work [52, 53], where an accumulator-oriented ISA imposes a one-consumer-only policy. Figure 9.8 compares these schemes to my CR-based approach on the 8x1w machine. (I show data for the 8x1w configuration because it is on the narrower clusters that effective collocation policies count most.) The results confirm the importance of proactively pushing unimportant work away and, furthermore, of doing so with more than just a local (greedy) strategy. Overall, both schemes add about 11% penalty to the CR-based

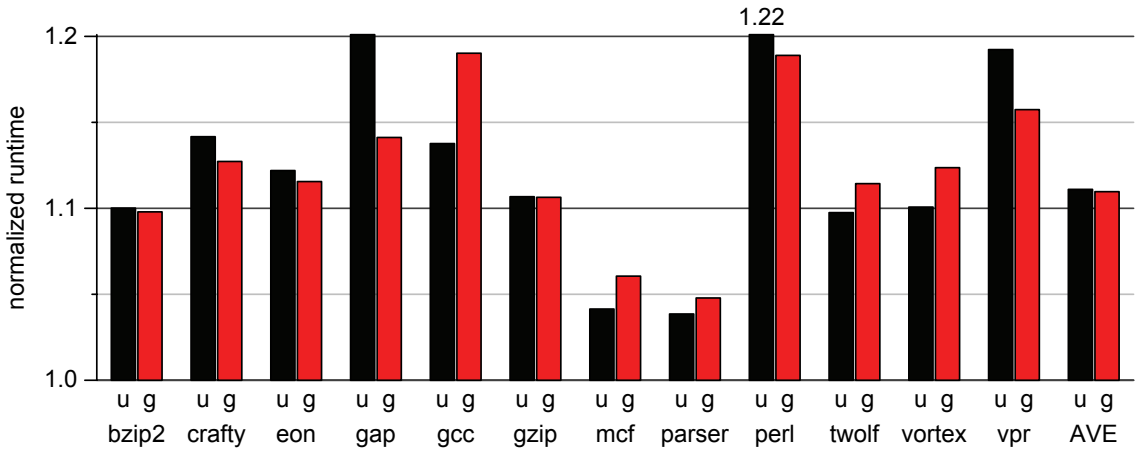


Figure 9.8. Policies that ignore CR values in aggregation. The left bar in each pair (those labeled 'u') shows an unrestrained scheme that permits all consumers to collocate with their producer. The right bars (labeled 'g') aggregates greedily by pushing all but the first consumer away. Data is for the 8x1w configuration, normalized to that for the CR-based collocation scheme on the same machine.

policy, reaching over 20% in the case of `perl`. Relative to the monolithic machine, this constitutes an average slowdown of approximately 20%, more than twice that incurred by the CR-based scheme.

9.6 Cleaning up singletons

In this section, I briefly revisit the logic of Algorithm 2, aiming to improve its *aggregation factor* — the extent to which it groups μ-ops into m-ops. I first show that the intra-block scope at which it operates, though a constraint on the achievable aggregation factor, is not the principal cause for singletons. Rather, the problem derives ultimately from the fact that CR-based aggregation, as it stands, only aggregates instructions in order to localize the dataflow between them. There is opportunity, nevertheless, to use aggregation also as a means for grouping (sometimes independent) instructions in order to aggregate load-balance decisions. That is, CR-based aggregation currently amplifies dependence-based steering decisions only, but load-balance decisions can likewise benefit. Given that the CR-based scheme has already proven itself to be very effective at finding the right instruction

collocations, the approach I take is to perform one or more *cleanup* passes after CR-based aggregation has been performed. In effect, I use CR-based aggregation to form a basic set of m-ops in each block, then patch the results by merging singletons, either with one another or into other, larger m-ops.

9.6.1 The intra-block constraint

Recall from Figure 9.6 that about 50% of all m-ops are singletons. Zooming in on those singletons, I find that about 25% of them have an above-threshold CR, but their preferred producer is not local to their basic block. That is, about one quarter of all singletons would like to collocate with a producer in a remote block. Clearly, there is opportunity for more aggressive collocation if m-ops would be permitted to span basic block boundaries. This would require incorporating into the aggregation process a region-formation scheme, for which purpose traces [36], superblocks [43], or even atomic regions [68, 79], could be used. In all these cases, static control flow speculation is necessarily embedded into m-op formation logic, a requirement that introduces some complexities into the co-design infrastructure. Though such issues are not insurmountable, they are beyond the scope of this preliminary study. Nevertheless, insight into the benefits (to aggregation factor) can be obtained by means of a simple experiment. Rather than statically constrain aggregation to a single basic block, I aggregate a fixed number of dynamically contiguous blocks. Though not perfect, this gives some idea of the extent to which region-based aggregation is liable to benefit m-op formation. Figure 9.9 shows the results of that experiment for the 4x2w machine. While it is clear that larger regions benefit aggregation, the improvement in the m-op size distribution is very modest. Indeed, average m-op size increases from 1.53 to only 1.84 as the aggregation region increases from 1 to 8 contiguous basic blocks. Though this result does not constitute an upper bound on the potential of region-based aggregation, it does indicate that increasing the aggregation scope will have very limited impact on the aggregation factor.

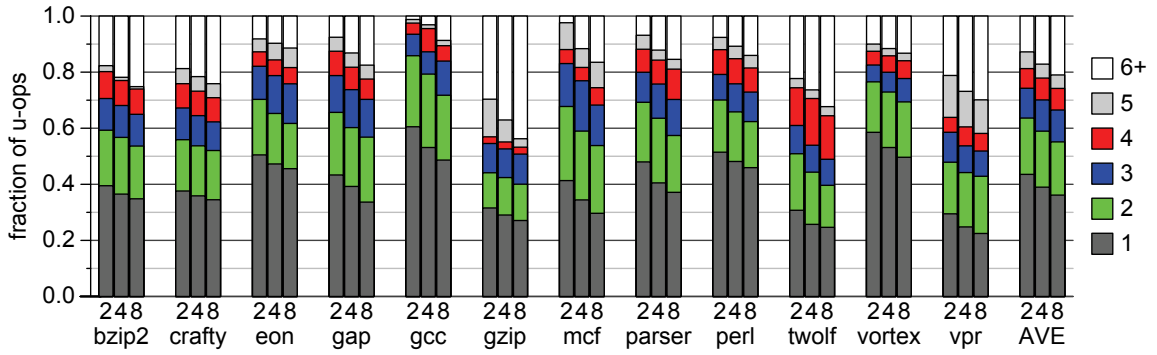


Figure 9.9. Distribution of m-op sizes for region-based aggregation. Like Figure 9.6, each bar shows the distribution of m-op sizes for the 4x2w machine configuration. Bars labeled '2', '4' and '8' show, respectively, the breakdowns for aggregation that spans 2, 4 and 8 dynamically contiguous basic blocks.

9.6.2 Merging singletons

If larger regions offer only limited benefit, it must be that the CR-based aggregation logic itself is the principal factor preventing a reduction in singleton count. The problem arises, ultimately, because the aggregation logic seeks only to group instructions if collocating their dataflow appears to be profitable. That is, amplification benefits to steering can be achieved only for dependence-based steering decisions. However, a large fraction of steering decisions are load-balance decisions, where dataflow has no role to play. An example of this effect appears in Figure 9.4, where instructions O and Q are proactively pushed away from N, resulting in the formation of two singletons.

In as much as grouping of instructions into a m-op implicitly encodes a set of dependence-based steering decisions, it too can be used to encode a set of load-balancing decisions. For example, instructions O and Q in Figure 9.4 could be merged into single m-op, itself flagged for load-balancing. Though this would force both those μ-ops to be load-balanced to the same cluster, thereby potentially introducing contention stalls, the net impact on performance is liable to be small because both instructions are, after all, non-critical. In short, merging unimportant singletons offers an opportunity to amplify steering bandwidth for load-balance decisions — exactly analogous to the amplification achieved for dependence-based steering.

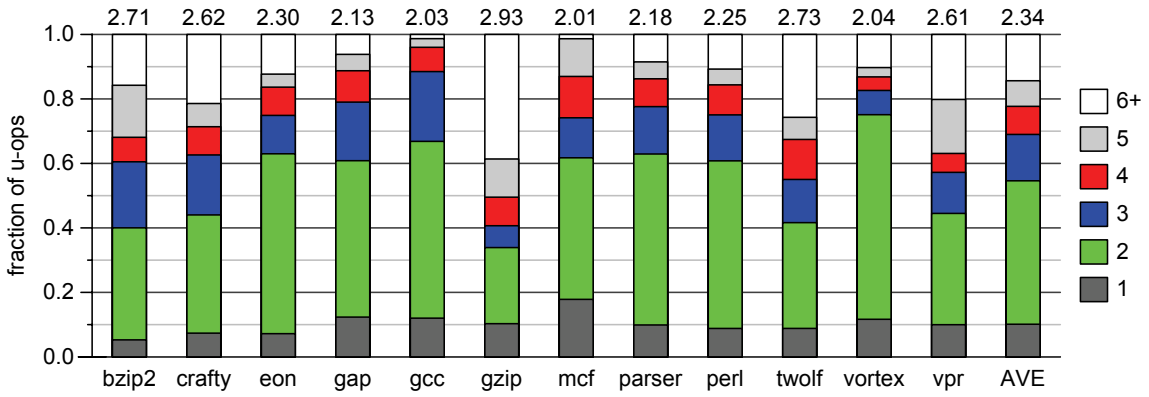


Figure 9.10. Distribution of m-op sizes after singleton cleanup. (*cf.* Figure 9.6.) Once again, data is for the 4x2w machine configuration, and the number at the top of each bar shows the average m-op size.

Since there are generally many choices in terms of which instructions ought to be paired, I adopt a policy that attempts to merge like-with-like in terms of LoC. This is because the LoC rating of a μ-op is determined, ultimately, by that of the most important μ-op with which it has been aggregated (Algorithm 2, line 13). Merging like-with-like dampens the extent to which low-LoC μ-ops are implicitly promoted in importance by virtue of being aggregated with more important μ-ops.

Figure 9.10 shows the distribution of m-op sizes resulting from a post-aggregation cleanup pass that pairs off independent, load-balance singletons, as described above. The overall distribution is clearly very different from that shown in Figure 9.6, with less than 10% of μ-ops now residing in singleton m-ops. This corresponds to an increase in average m-op size from 1.53 to 2.34, meaning that dynamic steering logic in the offline scheme is, on average, now making less than half as many decisions as is the logic in the online scheme. Of course, these benefits must be weighed against the potential for performance loss, which arises primarily as a result of contention introduced by more coarse-grained load balancing behavior. However, Figure 9.11 shows that the overall effect is marginal: the unshaded portions of bars labeled 's', which show the extra runtime incurred by the cleanup logic, are small or absent on most bars. Only the 8x1w machines show significant performance losses, particularly on benchmarks `gzip`, `parser` and `perl`. This is to be

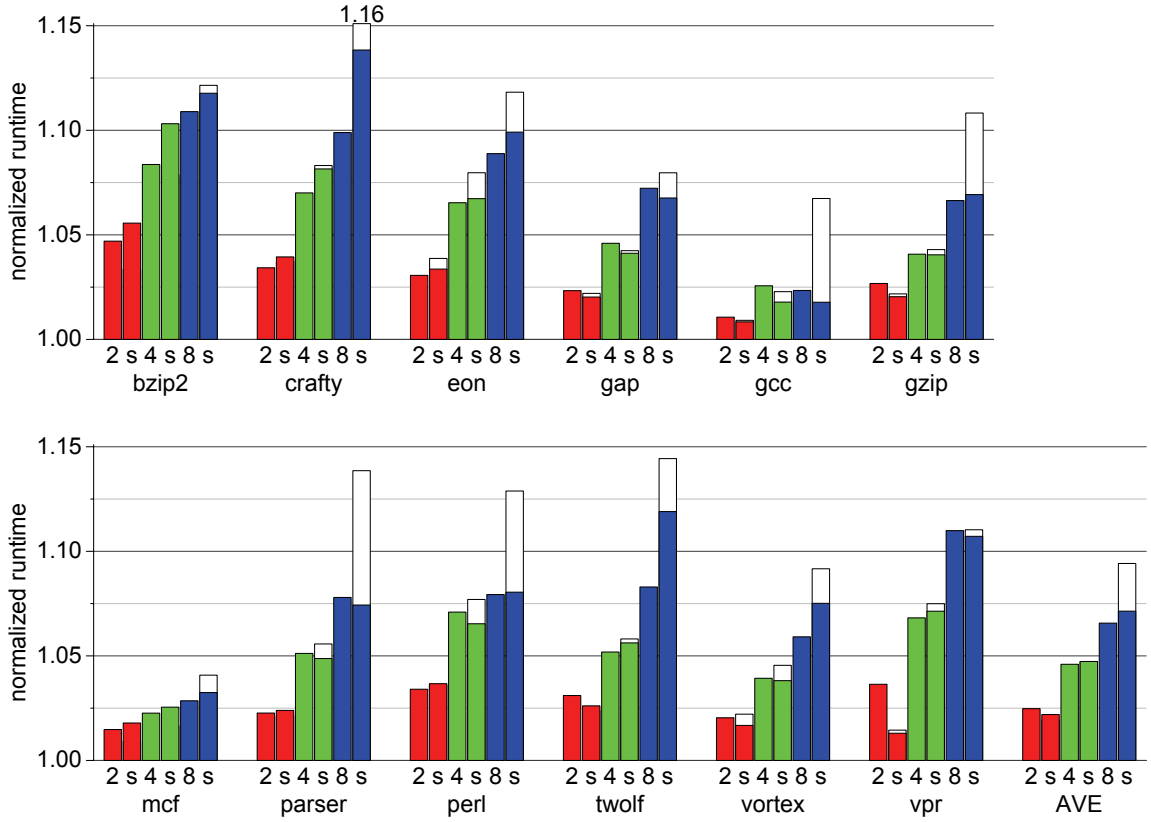


Figure 9.11. Performance after cleanup passes are applied. Like Figure 9.7, bars labeled '2', '4' and '8' show runtime of the dynamic LoC-based policies on the 2-, 4- and 8-cluster machines. The adjacent bars labeled 's' show runtime on the corresponding static machine: the shaded portion, which reproduces the data from Figure 9.7, corresponds to the basic CR-based aggregation logic; the unshaded portion shows runtime after the cleanup passes have been applied. LoC and CR data in these bars is derived from fabricated instruction traces.

expected, since the narrower clusters are very susceptible to resource contention problems, and these are exacerbated by cleanup logic. Though it is possible that a more judicious cleanup process would mitigate these problems, I feel that the narrow issue width on the 8x1w imposes a fundamental requirement for fine-grained load-balancing of instructions. This is clearly at odds with the coarse-grained decisions needed in order to achieve amplification benefits.

By no means do these results represent an upper bound on the potential of intra-block aggregation logic. Rather, they demonstrate that a relatively simple scheme is sufficient for more than halving the amount of steering decisions required of the dynamic steering logic. I believe a more sophisticated set of algorithms might well be able to improve intra-

block aggregation to the point that average m-op size reaches 3. Beyond that, however, the incremental benefit to hardware becomes increasingly small. Moreover, larger m-ops, especially those formed by cleanup logic, are liable to increase the number of spurious dispatch stalls that occur when large, unimportant m-ops cannot be dispatched in their entirety to a single cluster. Equally, too much collocation of independent instructions will tend to increase the number of contention stalls that appear on the critical path.

9.7 Summary and conclusions

In this chapter, I presented the first steps in a potentially very large study of a combined hardware/software scheme for managing the resources of a large, clustered machine. My objective has been to demonstrate the viability of such an approach, not to develop a detailed design. In this respect, the results presented in this chapter are very encouraging. I have shown that a relatively simple offline scheme, which involves the use of list scheduling of fabricated instruction traces to guide static steering decisions, is highly effective at discovering which instructions ought to be collocated and, equally, which ought not to be. The results of that analysis are communicated to dynamic steering logic by means of *instruction aggregation*, which groups instructions into larger entities called m-ops, the members of which are all steered to the same cluster. Dynamic steering logic, in turn, is vastly simplified relative to the logic relied upon in Chapters 7 and 8, being now reduced to a simple dependence-based policy with load-balancing capabilities. Moreover, it need make fewer decisions per cycle than the previous logic, since it benefits also from a bandwidth amplification effect introduced by instruction aggregation.

Though these results are promising, there is clearly much work to be done before any rigorous claims about the appeal of a co-designed machine can be made. To that end, I think there are two compelling directions for future work, both of which aim to enhance and extend the amplification benefits brought by instruction aggregation.

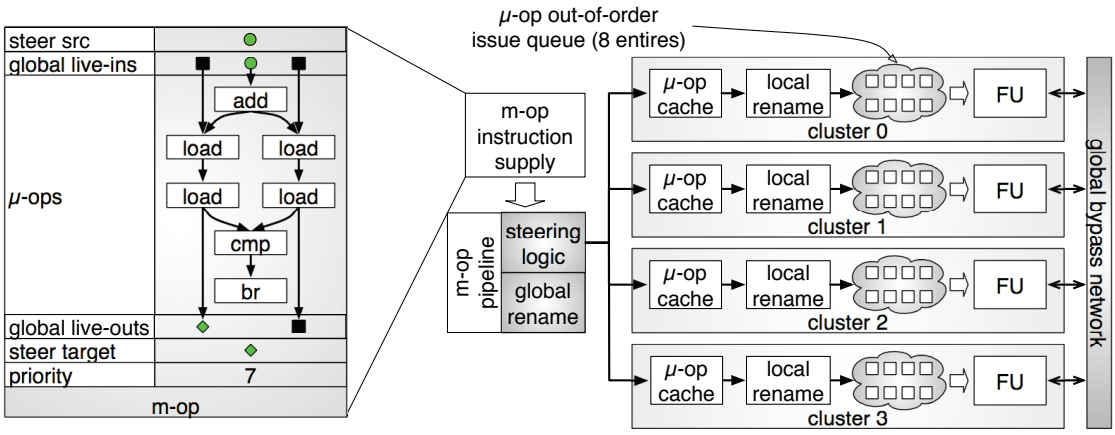


Figure 9.12. A machine treating m-ops as first-class entities. In contrast to the machine shown earlier in Figure 9.1, this machine has a front-end that operates exclusively on m-ops. When a m-op is steered to a cluster, its constituent μ-ops are fetched from that cluster's local μ-op cache, then flow down a short μ-op pipeline before entering the issue queue.

9.7.1 Macro-ops as first-class entities

Recalling Figure 9.1, the machine I have been simulating up to this point treats m-ops only as meta-data that accompanies μ-ops through the pipeline. A more compelling design is shown in Figure 9.12. This machine *decouples* the front-end from the execution core, the former being responsible for driving control flow at the coarse granularity of m-ops, and the latter for executing the μ-ops embedded within the m-ops being fetched. Viewed in this light, m-ops are more than just collocation groups: they are now self-contained macro-instructions.

Treating m-ops now as instructions in their own right permits a distinction to be made between dataflow that is *local* to a m-op, being communicated only among the constituent μ-ops, and *global* dataflow, which occurs between m-ops. The local values are those that are defined and consumed entirely within a single m-op. As such, they need not be written to the machine's architected state. They are, in a sense, merely temporary working storage needed to effect execution of the m-op. This leads to a notion of a *hierarchical namespace* for registers [52], where local names are used for values produced and consumed within each m-op (and hence cluster), and global names are used for values produced and con-

sumed across m-op boundaries. Only global values, which constitute live-in and live-out values at the m-op level, need be renamed in the front-end (hence the *global rename* logic shown in Figure 9.12), and only they need be written to the global register file once produced. Equally, local values need only be assigned register file storage at the cluster at which they will be used.

These changes will bring amplification benefits to both register rename and to the size and number of ports required of the register file storage. But new questions and challenges are now raised. First, it is not *a priori* clear what fraction of values can be classified as local — and hence to what extent register file and register rename amplification benefits will be felt. This will be a function of how program dataflow properties interact with the aggregation logic. Second, the number of global live-ins and live-outs that each m-op tends to have will be important for encoding and renaming purposes. Finally, since control flow is now to be driven at the granularity of m-ops, global dataflow among m-ops must constitute an acyclic dependence graph — or m-ops will not be serializable into an instruction stream. Overall, there is liable to be tension between aggregation decisions that effect good steering and the requirements for register file amplification and instruction stream serializability. Empirical exploration of these issues is needed to understand the various trade-offs.

9.7.2 Region-based aggregation

Though I showed that aggregation across control flow boundaries will, from an amplification point of view, have only limited benefit, the formation of regions larger than basic blocks would provide a flexible framework in which to explore more sophisticated aggregation algorithms. Single basic blocks are, in this respect, very restrictive. However, permitting m-ops to span control flow decisions introduces the potential for control flow to leave — and enter, if regions are formed as per trace scheduling [36] — the region over which instructions have been aggregated. That is, regions are necessarily speculative in nature, which introduces some potentially difficult problems. Any m-op that spans a control

flow point within the region must now deal with the potential that it executes only partially; or, that its entire execution be rolled back and partially reproduced on the exit path. And if m-ops are now to be viewed as self-contained instructions, with different local and global namespaces for register dataflow, then ostensibly local dataflow might be rendered global when static assumptions about control flow turn out to be wrong at runtime. For these reasons, I believe *atomic regions* [68, 79] are the most appealing method for forming regions over which instructions can be aggregated. These have the favorable property that static assumptions about behavior can be enforced in the code without having to explicitly deal with dynamic cases where assumptions turn out to be wrong: hardware permits all speculative execution within the region to be rolled back, and execution to resume on a less speculative version of the code. Support for such regions is already present in real machines like Transmeta's Crusoe [25], and has also been proposed in several research contexts [68, 79]. A very interesting study, therefore, would couple a realistic region formation scheme with the aggregation logic described earlier. The result, I believe, would be a realistic scheme for the formation of m-ops whose aggregation factor exceeds 3.

Chapter 10

Conclusion

In this concluding chapter, I discuss the insights gained and lessons learned from my study of a broad spectrum of machines that distribute instruction processing among disjoint execution units. Because the main technical results have already been enumerated in Chapter 1, I will only briefly repeat them here (Section 10.1). Rather, my objective now is to step back from the technical details and distill a set of general conclusions that can be drawn from this work as a whole. I enumerate these in Section 10.2. In Section 10.3, I then briefly review some of the main methodological techniques that distinguish this work from prior studies in ILDP. These, I think, could prove useful in a broader context than this work alone.

10.1 Summary

ILDP is a compelling design technique. By replicating small, simple structures, it offers a means for scaling execution core dimensions without, in the process, encountering the numerous power, clock and thermal problems that plague monolithic designs. This approach is appealing not only as a complexity-effective alternative to conventional designs, however. It will, I think, become increasingly important in the new CMP era as well, since TLP, alone, is not likely to provide substantial performance improvements for the large suite of programs that have, to date, resisted attempts at coarse-grained parallelization.

My objective in this work has been to understand, in a very general sense, the ramifications that a distributed mode of execution has on a machine's underlying ability to dynamically find and exploit ILP in the instruction stream. To that end, I adopted what

might be termed an *operational view*, focusing specifically on the *distributed* nature of instruction processing in an ILDP machine, and contrasting this with the *monolithic* mode of operation that characterizes conventional out-of-order designs. In this respect, my study stands in contrast to most prior work in this area, which has tended to focus mainly on microarchitectural innovation — for the purposes, specifically, of improving the clock, power or thermal aspects of a given design. My operational view permitted me to rather focus on the basic principles that affect the IPC capabilities of these machines, an area in which I feel a number of insights remain undocumented. Moreover, understanding the underlying capabilities of different distributed execution models facilitates a more informed evaluation of the trade-offs involved in balancing a given design's target IPC performance, power consumption, thermal output and clock rate.

In line with the operational view I take, I centered this work on the abstract notion of an *execution model* implemented by a given dynamic machine. I identified three such models: the dataflow-oriented one, which is implemented by conventional monolithic designs, and which serves as the ideal toward which ILDP machines strive; the distributed dataflow-oriented model, which is implemented by the clustered machines; and the slip-oriented model, implemented by the laned machines. Describing machine behavior in terms of these execution models permitted me to isolate the key *constraints* imposed by the two distributed models on instruction execution — constraints that are not present, or present only to a very limited extent, in the dataflow-oriented model of monolithic machines.

I find that the constraints introduced by distributed dataflow-oriented execution can be rendered largely benign; those in the slip-oriented execution model, however, impose fundamental bounds on the achievable performance. The latter problem arises because of program dataflow: ILP can be mined only from a very large number of simultaneously active dataflow chains. This very simple property has important ramifications for a machine that distributes execution among in-order execution units (lanes). Either the chains must be very carefully interleaved among the lanes (a burden that falls on instruction steer-

ing logic), or sufficiently many lanes must be provided to individually buffer all in-flight chains. I showed in Chapter 4 that neither option is appealing, the former because steering logic would perforce be more complex than conventional dynamic scheduling logic, the latter because a large number of lanes exacerbates the communication penalties that are inherent to a distributed design. In exposing these inherent limits, my results run counter to previous studies in this area, which have claimed optimistic performance figures for the laned machines. I showed in Chapter 3, however, that one of those studies overstated the potential of such designs owing to a simulation infrastructure that effectively hid the problems introduced by its slip-oriented execution core.

I want to emphasize that these results are negative only from the perspective of IPC performance potential. While my study makes it clear that laned machines cannot be expected to approach the IPC performance of resource-equivalent monolithic machines, they remain a compelling design point owing to their many implementation benefits. Exploring such trade-offs, however, is beyond the scope of this work.

From an IPC point of view, the clustered machines are far more compelling. I showed in Part II of this dissertation that instruction criticality — specifically, the preferential allotment of resources to the most performance-critical instructions — is a viable means for overcoming a distributed dataflow-oriented model's performance constraints. Much of that work is underpinned by a new criticality metric, *Likelihood of Criticality* (LoC), which I introduced in Chapter 7 to overcome problems inherent to previously proposed schemes that prioritize instructions based on a binary notion of criticality. I showed that steering and scheduling policies guided by LoC are capable of reducing performance losses relative to a monolithic machine to about 6% — the best figures published for clustered machines to date. I also showed that LoC is a stable property of programs, both within and across runs, a property that lends it to offline analysis. I described a framework to do just that in Chapter 8. This uses basic profile data to fabricate synthetic instruction traces, which, being representative of real execution traces, can be analyzed to discern accurate LoC profiles.

That static LoC values can be used to good effect in guiding the above-mentioned steering and scheduling policies means an online infrastructure for detecting and predicting criticality is superfluous. In Chapter 9, I further explored the potential to enlist assistance from offline analysis by examining a co-designed framework in which software and hardware share the burden of making steering and scheduling decisions. Though very much a preliminary study, my results are promising. I find that a large fraction of dynamic decisions, particularly for instruction steering, are statically stable. I still rely on hardware to make some dynamic steering and scheduling decisions, but the ability to encode some of them in binary will commensurately reduce the burden on — and, hence, the complexity of — the hardware. As I noted in Section 9.7, this co-designed infrastructure is an area in which I believe there are some potentially fruitful avenues for further research.

10.2 Principles of distributed execution

Stepping back, a number of very general conclusions can be distilled from this work. Many of them derive directly from the main results, some from the specific techniques I used to obtain those results. Overall, they constitute a set of principles that, I think, have bearing on any machine that distributes instruction processing among disjoint execution units. In fact, some, being applicable to dataflow itself, will almost certainly have at least some relevance for microarchitectural research in general.

10.2.1 Properties of dataflow

It is perhaps dangerous to talk in any general terms about *properties of dataflow*. Dynamic dataflow is far from homogeneous, both within and across programs. It also changes as compilers and programming practices evolve. Nevertheless, many of my results derive from generalizations of specific patterns and properties I have observed in SPEC CINT2000 programs. I am convinced, therefore, that general statements about dataflow, however in-

formally expressed they may be, can be very useful in guiding microarchitectural studies. Below, I summarize two general properties that I believe are most important.

Dataflow is amenable to distributed execution

The idealized list scheduling study of Chapter 6 proves that critical dataflow chains — in the SPEC CINT2000 programs, at least — tend to embed no ILP. That is, critical dataflow can generally execute without delay at even 1-wide PEs. The comparatively few cases in which critical chains do embed ILP arise because of *dataflow convergence*, where dyadic consumers obtain operands from two equally critical producers.

Convergence of critical dataflow chains poses a fundamental problem for any distributed execution model. First, even if PEs are wide enough to exploit the ILP available in converging chains, steering logic must have advance knowledge that convergence is imminent if it is to ensure that those (hitherto independent) chains are collocated early enough to avoid global communication penalties. Second, irrespective of advance knowledge, narrow PEs will necessarily impose a penalty on any critical dataflow that embeds more ILP than can be exploited locally: either the dataflow is partitioned across PEs and global communication penalties are incurred, or it is collocated at a single PE and resource contention stalls are incurred. Though (critical) convergence is very rare in the programs I studied, this is not a claim I can generalize to all code. For example, any programs that embed reduction-like dataflow patterns will, by definition, exhibit prevalent convergence. If all the chains involved in that convergence are critical (*i.e.* have no slack), a machine with narrow PEs will suffer significant performance losses. An extension of my idealized study to programs such as those in the SPEC CFP2000 suite would provide some answers to these questions.

Narrowness of (critical) dataflow is a *necessary* condition for efficient distributed execution. But it is not a *sufficient* condition: there remains a requirement that non-critical dataflow be able to shoulder penalties imposed on it in the process of ensuring the critical chains are not delayed. That is, non-critical dataflow must have sufficient slack. In

this respect, it is very difficult to draw general conclusions. Certainly in the case of distributed dataflow-oriented execution, the requirement holds: the idealized list scheduling study proves that non-critical dataflow, if judiciously allocated resources, is readily able to absorb penalties. Demonstrating the same for slip-oriented execution, even under idealized conditions, is much harder — not surprising because the penalties now incurred are much more severe than global communication and resource contention. Using an idealized *operation scheduling* framework [31], I have been able to show that certain configurations of laned machines can achieve IPC that is within about 5% of an equivalent monolithic machine. But this result demands a great deal of sophistication in the idealized scheduler: it must slot instructions based on a very careful balancing of the internal and external dimensions of the steering cost metric (as per Section 4.3), with precedence being given to one or the other dimension depending on the criticality of the instruction being steered relative to that of all instructions not yet slotted. I did not present these results in my work because, no matter how good slip-oriented execution can be shown to be under idealized assumptions, the problem of achieving even modest performance under realistic assumptions is demonstrably hard. In short, the question of idealized potential in the case of laned machines is moot.

Dataflow comprises many short chains

The underlying cause for the problems encountered by laned machines is that ILP does not present itself in a regular, neat manner in the instruction stream. Rather, it must be mined from a large number of independent and simultaneously active dataflow chains. I argued in Chapter 4 that the principal cause for this is non unit-latency instructions. In general, I find that an IPC of N must be extracted from between $2N$ and $3N$ independent chains.

A machine that supports dynamic scheduling is eminently suited to this requirement because it can easily find and issue any ready instruction from any one of those active chains — no matter where in the issue queue those chains may reside. In effect, it induces

a *temporal interleaving* of their execution. Doing likewise in a laned machine necessitates a *spatial interleaving* of those chains as well, because now the location of an instruction in the window determines when it can execute. It must therefore fall to steering logic to effect the right interleaving — to create an execution plan. But it must do so in program order, in a superscalar fashion, and well in advance of all instruction latencies being known. This is clearly too complex for any reasonable design. A dynamically scheduled machine circumvents these problems because it is able to buffer instructions without ramifications for when they can issue; and it issues instructions by *responding* to the current state of execution as opposed to *planning* it.

10.2.2 From performance in principle to performance in practice

The comparative ease with which the idealized scheduling results were obtained (Chapter 6) belies the complexity involved in achieving comparable results in practice (Chapter 7). It is instructive, therefore, to consider the main differences between the idealized and practical environments embodied by those two studies. Below, I compare them by considering the various concessions that must be made in moving from an idealized to a practical setting. In the process, I expose a number of underlying principles at work in distributed execution models.

Criticality

In addition to proving that program dataflow is indeed amenable to being processed in a distributed manner, the idealized study demonstrated that instruction criticality has a central role to play in orchestrating that processing. But the criticality information relied upon by the idealized scheduler, which derives from perfect "upstream knowledge", is not available in a realistic setting. A practical scheme must instead rely on past program behavior as in indicator of the future; it must *predict* instruction criticality. Unfortunately, accurately identifying which dynamic instructions are going to be critical is fundamentally hard, in

the sense that an ability to do so would amount to knowing exactly which branches will be mispredicted and which loads will miss in the cache. Moreover, even perfect knowledge of the critical path is not sufficient. What is ultimately needed is an ability to correctly *prioritize* between any two instructions, be they critical or not. The *Likelihood of Criticality* metric is ideally suited to this purpose because, in a sense, it reflects the expected cost of making the wrong choice. For example, an instruction with an LoC of 75% will add 1.5 cycles to runtime if a 2-cycle global communication penalty is imposed on each of its dynamic instances; an instruction with an LoC of 25% will add about 0.5 cycles per instance if likewise penalized. If prioritization logic consistently prefers the former over the latter, there is a expected saving of 1 cycle per dynamic instance.

LoC is effective because it avoids attempting to make an explicit prediction of a dynamic event. It merely records the past tendency of that event to occur. Binary prediction schemes, by contrast, *aggregate* an event's past behavior into a single, definitive prediction about its next occurrence. In the process, they lose information because they reduce a wide variety of different dynamic behaviors to a single binary value. The ability to gauge *relative* criticality is thereby lost. An analogous observation was recently made by Malik *et al.*, who showed that conventional *path confidence predictors*, in classifying branches as either high- or low-confidence, effectively lose valuable information about the *relative confidence* of different branch predictions [60]. Those relative differences turn out to be very important when they are combined (multiplied) to obtain an estimate of overall path confidence.

Local versus global optima

In combining instruction steering and scheduling into a single step, and in making its decisions only when instructions become data-ready, the idealized scheduler effectively implements an out-of-order steering policy. In the process, it gains the ability to make decisions always with a *global optimum* in mind: among the data-ready instructions, it gives precedence to those that are most likely to affect the whole region's runtime, allowing them

first choice in terms of where they will be slotted (*i.e.* steered); the less important instructions are slotted "in the gaps", as and when a lack of more critical instructions permits. A realistic steering policy must of course operate in-order, since its decisions are made in the front-end of the pipeline. Unfortunately, there will, in general, be no relationship between ordering of instructions based on their priority and their ordering in the instruction stream. Thus, any steering policy that operates in a *greedy* fashion will necessarily produce sub-optimal results. Steering must therefore be cognizant of the problems posed by *local optima*: it is not always globally beneficial to slot an instruction into its preferred location. The *proactive load balancing* policy I discussed in Section 7.4 aims to avoid precisely this problem.

This issue of local versus global optima is important because it imposes a difficult requirement on hardware. In the absence of a global view of future dataflow, it is hard to know when a globally sub-optimal decision is about to be made. That said, it is precisely cases like this that lend themselves to offline analysis. Software is ideally placed to exploit a more global view, permitting it to more effectively make globally optimal decisions. Indeed, that the static policies I explored in Chapter 9 were sometimes able to outperform their dynamic counterparts is a result, I believe, of software's ability to do just that.

Shelving

Another problem arising from having to perform steering in-order relates to *buffering*. In slotting instructions only when they become data-ready, the idealized scheduler implicitly buffers those whose operands are still pending. This capability isolates the scheduler from an important practical constraint: instructions in a real machine can only be buffered *after* a steering decision has been made. Put another way, without an ability to buffer instructions, steering logic in a real machine is exposed to a new problem: there is now the possibility that buffer resources at the desired steering target will not be available, a situation not entirely unlikely given the comparatively small issue queues at each PE. When this happens,

steering logic is faced with a choice: either it *stalls* to wait for the preferred PE to become available, or it *steers* the instruction to another (available) PE. Previously published policies have adopted one or the other approach. Some, like the dependence-based policy from Palacharla *et al.* always stall if the preferred target is not available; others, like the various load-balancing schemes proposed for clustered machines, always steer to another PE. Neither approach is uniformly correct: sometimes it is better to stall, sometimes to steer.

The need for both types of behavior is explained by the notions of fetch- and execute-criticality. In fetch-critical regions, the sole concern is sustaining forward progress by rapidly dispatching instructions into the window, even if this must be done at the cost of imposing execution delays on those instructions. Execute-critical regions, by contrast, demand steering decisions that judiciously avoid execution delays, a requirement that frequently necessitates sacrificing forward progress. Overall, steering logic must manage a trade-off between finding the optimal target PE for an instruction (which may not be possible at the current time) and sustaining forward progress to (perhaps more critical) instructions that lie ahead. I showed in Chapter 7 that LoC is an effective tool for managing this trade-off.

It is important to understand that stalling the instruction supply in execute-critical code regions does not penalize performance — the instruction supply rate is not the bottleneck. Thus, a monolithic machine's ability to buffer fetched instructions in such regions is superfluous: it serves merely to deal with an over-supply of instructions, not as a means for exposing ILP to the issue logic. But this buffering is, at the same time, benign. In holding an instruction in the issue queue, the monolithic (dataflow-oriented) machine imposes no constraint on the ability of that instruction to subsequently execute. Not so in a distributed machine, where buffering an instruction now has ramifications for its execution (because of the potential for global communication and resource contention). This change exposes an important difference between the monolithic and distributed execution models. The former's very flexible buffering capability, which has previously been referred to more aptly

as *shelving* [89], achieves a degree of *decoupling* between instruction supply and the order and rate at which instructions execute. Both the front-end and execution core operate independently of one another, the former always dispatching instructions into the window as fast as possible and the latter executing them as per their dataflow dependences. In distributed execution models, by contrast, the front-end and execution core are somewhat coupled, the former having now to be cognizant of the latter's ability to keep up.

10.3 Methodology

I have made use of a number of tools and techniques in this work that have proven themselves to be enormously useful in understanding, and in gaining new insights into, the operation of dynamic machines. As such, they warrant specific mention. Moreover, I believe their use distinguishes my work from most other studies in ILDP (and in microarchitectural studies in general), where the sole empirical tool tends to be simulation.

10.3.1 Conceptual models

I have frequently made use of a number of high-level, abstract models (not all of which are mine) to reason about the behavior of a dynamic machine, be it monolithic or distributed.

1. *Execution models.* The monolithic and distributed dataflow-oriented execution models, and the slip-oriented execution model, provided the means for succinctly describing and delineating the microarchitectural landscape to which this dissertation is devoted. They also proved useful as a means for distilling the key features that distinguish different types of distributed machines, as well as for isolating the factors that have the most bearing on their performance.

2. *Dependence graph model of criticality.* The framework developed by Fields *et al.* pervades my work. In capturing both the dataflow and microarchitectural constraints,

this explicitly takes into account their combined role in determining overall performance. All of my results are built on this view of machine behavior, and I firmly believe that other studies could likewise benefit.

3. *Interval analysis.* This technique has previously been used to very good effect in the development of abstract models of performance [30, 48, 65], and even as the basis for a hardware profiling proposal [29]. Partitioning an instruction trace into regions, and reasoning about each in isolation from the others, was useful in my own work both as an empirical tool (in the idealized scheduler study) and as a conceptual one (for understanding, for example, the notions of fetch- and execute-critical code regions).

4. *Matrix model.* This abstract view of the operation of laned machines is one born of my frustrated attempts at trying to find a good steering policy for laned machines. I used it initially as a conceptual framework in which to reason about steering. It subsequently became the platform on which I constructed an argument about the inherent complexity of good steering policies.

I enumerate these models not only for their own sake, but also to distill a more general point about methodology. Throughout my work, I have tended to approach problems *from the perspective of* a high-level, usually informal, model of machine operation. Those models are always a gross simplification of reality, so they are not, by themselves, very useful. Rather, they serve as a tool — a conceptual framework — in which to ground one's reasoning about very detailed, very complex problems. This is a technique that has proved indispensable in my study of ILDP, and can almost certainly be likewise useful in other areas. Moreover, I have found the models themselves, being simple and concise expressions of a few key ideas, serve very well as a vehicle for communicating the insights derived from them.

10.3.2 Idealized studies

When I started my exploration of ILDP, it was with the preconceived idea that distributing execution among small processing elements is essentially a means for trading IPC performance for implementation benefits. That is, the appeal of ILDP lay principally in its potential to benefit the power, clock and thermal aspects of a design, IPC penalties notwithstanding. But this view that IPC must necessarily be sacrificed, which pervades all prior work in this area, was born not of any conclusive evidence that distributed execution is *inherently* limited, but rather of a failure on the part of previous studies to demonstrate good IPC performance. From my perspective, then, there remained open an important question about inherent performance potential versus performance achieved in practice: *are previously reported IPC figures merely artifacts of ineffective use of the underlying hardware, or do they reflect a more fundamental problem introduced by a distributed execution core?* I believed then — and still do — that knowing the inherent performance potential of a machine (or class of machine) is of central importance in any study, both in principle (because we simply ought to know) and in practice (because it either justifies or renders pointless any further research in the area).

My attempts to discover best-case performance potential have led to important results in both the laned and clustered machine categories. In the former, it was my failure to obtain good results that led me develop the matrix model for reasoning about slip-oriented execution. In the latter, the comparative ease with which good results could be produced motivated and justified the search for an effective set of practical policies. In both cases, these good results were facilitated by the ability to reason about the problem without having to deal with the artifacts of specific implementation mechanisms. This permits distilling the key concepts — the internal/external cost trade-off in the case of laned machines; the utility of giving preferential treatment to critical chains in the case of the clustered machines. I firmly believe that it is through an understanding of the basic principles that practical results are best sought. And those principles are best learned by means of idealized studies.

10.3.3 Critical path analysis

I have made extensive use in this work of postmortem critical path analysis — or *focused analysis*, as I called it in Chapter 1 — to characterize and diagnose problems with the performance of various simulated machines. For example, I used it to identify the underlying sources of performance loss in the dependence-based scheduler from Palacharla *et al.* (Chapter 3) and the focused steering and scheduling policies from Fields *et al.* (Chapter 7). Underpinning those results is the *critical path signature*, which provides a concise, quantified breakdown of the various factors contributing to overall runtime. I used those breakdowns to *zoom in* on specific aspects of behavior that are clearly causing performance problems. Had I instead relied on aggregate metrics (*e.g.* total global communication penalties incurred or total contention stalls), I would likely have ended up wasting time on exploring irrelevant factors, or, worse still, developing schemes that target aspects of behavior that have little or no bearing on performance.

A critical path signature is also useful simply as a tool for verifying that a given design is *balanced*, in the sense that it does not spend an inordinate amount of its time on any one aspect of performance — or, equally, that it is indeed limited by the factors it is expected to be limited by (*e.g.* `mcf` ought to observe a large contribution from memory latency). It is also useful for understanding the often subtle effects that microarchitectural changes have on overall performance. For example, I used the critical path signature as a vehicle for understanding how different microarchitectural parameters changed the apparent efficacy of dependence-based scheduling (Section 3.3). Finally, I have even used the critical path signature as an aid in debugging simulator code. Often subtle errors in timing information are exposed when the various buckets in a critical path breakdown do not add up to overall runtime.

References

[1] J. Abella and A. González. Inherently workload-balanced clustered microarchitecture. In *Proceeedings of the 19th International Parallel and Distributed Processing Symposium*, page 20a, April 2005.

[2] Advanced Micro Devices, Incorporated. Lightweight profiling proposal. `http://developer.amd.com/assets/HardwareExtensionsforLightweightProfilingPublic20070720.pdf`, August 2007.

[3] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 205–216, December 2003.

[4] AGEIA Technologies, Inc. PhysX by ageia. `http://www.ageia.com/pdf/ds_product_overview.pdf`

[5] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 168–179, June 2001.

[6] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 275–286, June 2003.

[7] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 134–148, January 1998.

[8] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 337–347, December 2000.

[9] P. Bannon and J. Keller. Internal architecture of Alpha 21164 microprocessor. In *Proceedings of the 40th IEEE Computer Society International Conference*, pages 79–87, March 1995.

[10] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W.-M. Hwu. Beating in-order stalls with "Flea-flicker" two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 387–398, December 2003.

[11] R. Barnes, S. Ryoo, and W.-M. Hwu. "Flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 319–330, November 2005.

[12] R. Bhargava and L. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 264–274, June 2003.

[13] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[14] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 18–29, December 2004.

[15] A. Bracy and A. Roth. Serialization-aware mini-graphs: Performance with fewer resources. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 171–184, December 2006.

[16] D. Burger, T. Austin, and S. Bennet. Evaluating future microprocessors: the SimpleScalar tool set. Technical Report 1308, University of Wisconsin-Madison Computer Sciences Department, 1996.

[17] A. Buyuktosunoglu, A. El-Moursy, and D. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *Proceedings of the 15th International ASIC/SOC Conference*, pages 31–35, September 2002.

[18] R. Canal, J.-M. Parcerisa, and A. González. A cost-effective clustered architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 160–168, October 1999.

[19] R. Canal, J.-M. Parcerisa, and A. González. Dynamic cluster assignment mechanisms. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 133–142, January 2000.

[20] R. Canal, J.-M. Parcerisa, and A. González. Dynamic code partitioning for clustered architectures. *International Journal of Parallel Programming*, 29(1):59–79, February 2001.

[21] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998.

[22] ClearSpeed Technology. Whitepaper: CSX processor architecture. `http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf`, February 2007.

[23] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, October 1987.

[24] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 292–303, December 1997.

[25] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 15–24, March 2003.

[26] D. Dobberpuhl, R. Witek, R. Allmon, R. Anglin, D. Bertucci, S. Britton, L. Chao, R. Conrad, D. Dever, B. Gieseke, S. Hassoun, G. Hoeppner, K. Kuchler, M. Ladd, B. Leary, L. Madden, E. McLellan, D. Meyer, J. Montanaro, D. Priore, V. Rajagopalan, S. Samudrala, and S. Santhanam. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[27] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Department of Computer Science, Yale University, April 1986.

[28] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 253–262, June 2003.

[29] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–184, October 2006.

[30] S. Eyerman, J. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 48–58, March 2006.

[31] P. Faraboschi, J. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638–1659, November 2001.

[32] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, December 1997.

[33] B. Fields, R. Bodik, M. Hill, and C. Newburn. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, May 2002.

[34] B. Fields, R. Bodik, M. Hill, and C. Newburn. Using interaction cost for microarchitectural bottleneck analysis. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 228–242, December 2003.

[35] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[36] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 38(12):478–490, December 1989.

[37] M. Flynn, P. Hung, and K. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(2):11–22, March 1999.

[38] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[39] J. González, F. Latorre, and A. González. Cache organizations for clustered microarchitectures. In *Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 46–55, June 2004.

[40] M. Hill and M. Marty. Amdahl's Law in the multicore era. To appear: IEEE Computer, 2008.

[41] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[42] S. Hu, I. Kim, M. Lipasti, and J. Smith. An approach for implementing efficient superscalar CISC processors. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 41–52, February 2006.

[43] W.-M. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, May 1993.

[44] E. Ipek, M. Kirman, N. Kirman, and J. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, June 2007.

[45] N. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[46] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.

[47] K. Kailas, A. Agrawala, and K. Ebcioglu. CARS: A new code generation framework for clustered ILP processors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 133–143, January 2001.

[48] T. Karkhanis and J. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338–349, June 2004.

[49] G. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 239–246, August 1996.

[50] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[51] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. Keckler. Composable lightweight processors. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 381–393, December 2007.

[52] H.-S. Kim and J. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 71–82, May 2002.

[53] H.-S. Kim and J. Smith. Dynamic binary translation for accumulator oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 25–35, March 2003.

[54] I. Kim and M. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 277–288, December 2003.

[55] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, December 2003.

[56] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 64–75, June 2004.

[57] M. Lam and R. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

[58] D. Landskov, S. Davidson, B. Shriver, and P. Mallett. Local microcode compaction techniques. *ACM Computing Surveys*, 12(3):261–294, September 1980.

[59] J. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1):51–142, May 1993.

[60] K. Malik, M. Agarwal, V. Dhar, and M. Frank. PaCo: Probability-based path confidence prediction. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 50–61, February 2008.

[61] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[62] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, March 2003.

[63] A. Mericas. The PowerPC performance monitor. In *Workshop on Hardware Performance Monitor Design and Functionality*, February 2005.

[64] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 27–36, January 2001.

[65] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 2–10, October 1999.

[66] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[67] G. Muthler, D. Crowe, S. Patel, and S. Lumetta. Instruction fetch deferral using static slack. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 51–61, November 2002.

[68] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 174–185, June 2007.

[69] N. Nihdi. Performance monitoring on Pentium 4 processors. In *Workshop on Hardware Performance Monitor Design and Functionality*, February 2005.

[70] D. Noonburg and J. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 52–62, November 1994.

[71] S. Nussbaum and J. Smith. Modeling superscalar processors via statistical simulation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, September 2001.

[72] E. Nystrom and A. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 103–114, November 1998.

[73] E. Öezer, S. Banerjia, and T. Conte. Unified Assign and Schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 308–315, November 1998.

[74] M. Oskin, F. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 71–82, June 2000.

[75] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin–Madison, 1998.

[76] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[77] S. Palacharla and J. Smith. Decoupling integer execution in superscalar processors. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 285–290, November 1995.

[78] J.-M. Parcerisa and A. González. Reducing wire delay through value prediction. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 317–326, December 2000.

[79] S. Patel and S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):300–318, June 2001.

[80] F. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. Keynote address at the 32nd International Symposium on Microarchitecture, December 1999.

[81] P. Racunas and Y. Patt. Partitioned first-level cache design for clustered microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 22–31, June 2003.

[82] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and stratification. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 110–120, February 2006.

[83] P. Salverda, C. Tucker, and C. Zilles. Accurate critical path analysis via random trace construction. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages xxx–yyy, April 2008.

[84] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 55–66, November 2005.

[85] P. Salverda and C. Zilles. Dependence-based scheduling revisited: A tale of two baselines. In *6th Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2007.

[86] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pages 252–263, February 2008.

[87] P. Sassone and D. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 7–17, December 2004.

[88] P. Sassone, D. Wills, and G. Loh. Static strands: Safely exposing dependence chains for increasing embedded power efficiency. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*, pages 127–136, June 2005.

[89] D. Sima. Superscalar instruction issue. *IEEE Micro*, 17(5):28–39, September 1997.

[90] J. Smith. Instruction-level distributed processing. *IEEE Computer*, 34(4):59–65, April 2001.

[91] J. Smith, S. Sastry, T.Heil, and T.Bezenek. Achieving high performance via co-designed virtual machines. In *Proceedings of the International Workshop on Innovative Architecture*, pages 77–82, October 1998.

[92] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, July 2002.

[93] S. Srinivasan, R. Dz-Ching Ju, A. Lebeck, and C. Wilkerson. Locality vs criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, July 2001.

[94] Standard Performance Evaluation Corporation (SPEC). CINT2000 (Integer Component of SPEC CPU2000). `http://www.spec.org/cpu/CINT2000`, September 2003.

[95] J. Stark, M. Brown, and Y. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 57–66, December 2000.

[96] Sun Microsystems. Throughput computing: Changing the economics and ecology of the data center with innovative SPARC technology. `http://www.sun.com/processors/whitepapers/throughput_whitepaper.pdf`, November 2005.

[97] D. Sylvester and K. Keutzer. Rethinking deep-submicron circuit design. *IEEE Computer*, 32(11):25–33, November 1999.

[98] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 185–195, January 2001.

[99] E. Tune, D. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.

[100] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[101] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.

[102] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 25–36, February 2007.

Author's Biography

Pierre Salverda earned his Bachelor of Science with Honours (1996) and Master of Science degrees (1998), both in Computer Science, at the University of the Witwatersrand, Johannesburg. During the course of those studies he was awarded several academic honours, including the Raikes Scholarship, the Altec Systems Prize for top fourth year research project, and the Liberty Life Gold Medal for outstanding achievement in the fourth year of study. His Master's research, entitled "An SRAM Main Memory Model" was nominated for the best Master's research project award in 1997. Its main results were presented at the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

After completing his studies, he worked as a software developer in the financial industry in London until 2002. He then returned to academia, commencing his PhD studies in 2002 at the University of Illinois at Urbana-Champaign. For his doctoral research he investigated Instruction-Level Distributed Processing, focusing, in particular, on the interplay of program dataflow and microarchitectural constraints in determining the overall performance of such designs. Parts of his dissertation work were published at the International Symposium on Microarchitecture (2005), the International Symposium on High-Performance Computer Architecture (2008) and the International Symposium on Code Generation and Optimization (2008). During the course of his studies, he was also involved in other research efforts investigating formal methods and their use in microarchitectural verification. Part of that work was published at the Formal Methods Europe conference in 2005.